Express

Fast • Minimal • Flexible

Express.js Basics

— • Node.js Framework for Web Applications • —

1. Express.js Logo

Express.js logo represents a fast, unopinionated and minimal framework.

Express

2. Node.js Framework

Built on Node.js, Express.js is a powerful framework for building web applications.

node

3. Fast

Express.js is designed for speed and high performance.

4. Minimal

Minimal features set. Only what you need, nothing extra.

5. Web Server

Create robust web servers to handle HTTP requests and responses easily.

6. REST API

Perfect for building RESTful APIs quickly and efficiently.

`GET /api/users`

7. Middleware

Use middleware functions to handle requests, responses, and next steps.

Request → Middleware → Response

8. Routing

Define routes for different endpoints and HTTP methods with ease.

`GET /about` | `POST /login`

Hello World with Express

```
1  const express = require('express');
2  const app = express();
3
4  app.get('/', (req, res) => {
5    res.send('Hello from Express.js!');
6  });
7
8  app.listen(3000, () => console.log('Server running on port 3000'));
```

Fast
High performance and non-blocking I/O.

Flexible
Unopinionated. Use the tools you love.

Developer Friendly
Simple API and easy to learn.

Production Ready
Used by millions of developers worldwide.

Express.js is the de-facto standard server framework for Node.js ♥

Express

Fast • Minimal • Flexible
Node.js Web Framework

EXPRESS.JS INSTALLATION & SETUP

Step-by-Step Guide to Get Started with Express.js

1 Initialize Project

```
$ npm init -y
```

Creates a package.json file with default values.

package.json

2 Install Express

```
$ npm install express
```

Installs Express.js and adds it to dependencies.

✓ Installed Successfully

3 Project Structure

- my-express-app/
  - node_modules/
  - package.json
  - app.js
  - server.js

4 Run with Nodemon

```
$ npm install -D nodemon
```

Nodemon automatically restarts your server on file changes.

5 Environment Variable

```
PORT=3000
```

Set PORT in .env file or your environment.

.env

6 Start the Server

```
$ npm start
```

Runs the server using the start script from package.json

package.json

```
{   "name": "my-express-app",
    "version": "1.0.0",
    "main": "server.js",
    "scripts": {
       "start": "node server.js",
       "dev": "nodemon server.js"
    },
    "dependencies": {
       "express": "^4.18.2"
    },
    "devDependencies": {
       "nodemon": "^3.0.1"
}   }
```

app.js

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello from Express!');
});

module.exports = app;
```

server.js

```
require('dotenv').config();
const app = require('./app');

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
   console.log(' Server running on
    port ${PORT}');
});
```

Run Commands

1	npm init -y	Initialize new project
2	npm install express	Install Express
3	npm install -D nodemon	Install Nodemon (dev)
4	npm run dev	Run with Nodemon
5	npm start	Start the server

What You Get

- Express.js installed
- Project ready
- Auto restart with Nodemon
- Environment configuration

Pro Tip

Use nodemon during development and remove it in production.

Happy Coding! 🚀

Default Port

3000

Change anytime using PORT variable.

Express

EXPRESS APPLICATION

Building Blocks of an Express.js Application

```
const express = require('express');
const app = express();

app.listen(3000, () => {
  console.log('Server running');
});
```

1 express()

Creates an Express application.

```
const express = require('express');
const app = express();
```

This is the first step to create any Express app.

2 app.listen()

Starts the server and listens on a port.

```
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log('Server running on port ${PORT}');
});
```

Starts the server and makes your app available on the given port.

3 app.use()

Uses middleware functions.

```
app.use(express.json());
app.use(express.urlonceded({ extended: true }));
app.use('/api', apiRouter);
```

Middleware runs for every matching request before the route handler.

4 app.get()

Handles GET requests.

```
app.get('/', (req, res) => {
  res.send('Hello from GET request');
});
```

Used to get data from the server.

5 app.post()

Handles POST requests.

```
app.post('/users', (req, res) => {
  const { name } = req.body;
  res.status(201).json({ message: 'User
  ${name} created' });
});
```

Used to send data to the server.

6 Request

Contains data sent by the client to the server.

req.params	Route parameters
req.query	Query string parameters
req.body	Data sent in request body
req.headers	Request headers
req.method	HTTP method (GET, POST, etc.)
req.url	Requested URL

7 Response

Contains data sent by the server to the client.

res.send()	Send text/HTML
res.json()	Send JSON data
res.status()	Set status code
res.redirect()	Redirect to another URL
res.end()	End the response
res.header()	Set response headers

8 Server

Handles requests and sends responses.

Express Server

Example: Complete Express Application

```
1  const express = require('express');
2  const app = express();
3  const PORT = process.env.PORT || 3000;
4
5  app.use(express.json());
6
7  app.get('/', (req, res) => {
8    res.send('Welcome to Express App!');
9  });
11 app.post('/users', (req, res) => {
12   const { name } = req.body;
13   res.status(201).json({ message:  'User ${name} created' });
14 });
16 app.listen(PORT, () => {
17   console.log('Server running on port ${PORT}');
18 });
```

How Express Handles a Request

Client → Request (HTTP Request) → Express Server (Routes & Middleware) → Response (HTTP Response) → Client

Key Points

- ✓ Express simplifies handling HTTP requests and responses.
- ✓ Middleware (app.use) is executed before route handlers.
- ✓ Routing methods (app.get, app.post) define endpoints.
- ✓ res object is used to send responses back to the client.

node JS

Express

EXPRESS.JS ROUTING BASICS

Routing is used to define how an application responds to a client request at a particular endpoint.

• Define Endpoints • Handle Requests • Send Responses •

HTTP METHODS

GET — Retrieve data from the server. — Read

POST — Send data to the server to create a resource. — Create

PUT — Update an existing resource completely. — Update (Replace)

PATCH — Update an existing resource partially. — Update (Partial)

DELETE — Delete a resource from the server. — Delete

COMMON ROUTE EXAMPLES

METHOD	ENDPOINT	DESCRIPTION	EXAMPLE
GET	/	Home route	`app.get('/', (req, res) => { ... })`
GET	/users	Get all users	`app.get('/users', (req, res) => { ... })`
GET	/users/:id	Get user by ID	`app.get('/users/:id', (req, res) => { ... })`
POST	/users	Create a new user	`app.post('/users', (req, res) => { ... })`
PUT	/users/:id	Update user (replace)	`app.put('/users/:id', (req, res) => {...})`
PATCH	/users/:id	Update user (partial)	`app.patch('/users/:id', (req, res) => {...})`
DELETE	/users/:id	Delete user	`app.delete('/users/:id', (req, res) => {...})`

BASIC ROUTING EXAMPLE

JS

```
const express = require('express');
const app = express();

// GET request
app.get('/', (req, res) => {
  res.send('Welcome to Express!');
});

// POST request
app.post('/users', (req, res) => {
  res.send('User created!'');
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

USING EXPRESS ROUTER

userRoutes.js

```
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
  res.send('Get all users');
});

router.post('/', (req, res) => {
  res.send('Create user!');
});

module.exports = router;
```

→ Router →

app.js

```
const express = require('express');
const app = express();
const userRoutes = require('./routes/userRoutes');

app.use(express.json());
app.use('/users', userRoutes);

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

Router helps to organize routes in separate files and keeps your code clean and scalable.

KEY CONCEPTS

Route
A route is a combination of HTTP method and endpoint that the server listens to.

Endpoint
The URL path where the client sends the request (e.g., /users, /users/:id)

Router
Express.Router() is used to create modular, mountable route handlers.

Middleware
Functions that run before route handlers to process requests and responses.

Express

EXPRESS.JS ROUTE PARAMETERS

Route Parameters let you capture values from the URL and use them in your routes.

Capture Dynamic Values in the URL

```
const express = require('express');
const app = express();

app.listen(3000, () => {
  console.log('Server running
  on port 3000');
});
```

1 :req.params

Access route parameters using req.params.

```
app.get('/users/:id', (req, res) => {
  console.log(req.params.id);
});
```

Output: Value of :id

2 :id

:id is a parameter in the route.

Example URL

GET /users/123

req.params.id → "123"

3 :userId

You can use any name for the parameter.

Example URL

GET /users/456/profile

req.params.userId → "456"

4 Slug

Capture slugs or unique strings in the URL.

Example URL

GET /blog/express-js-basics

req.params.slug → "express-js-basics"

5 Dynamic Route

Route changes based on the parameter value.

Example Routes

/products/1

/products/2

6 URL Parameter

Parameters are part of the URL path.

General Syntax

/resource/:parameter

7 Route Match

Route matches when the URL fits the pattern.

Example

Route: /users/:id

URL: /users/789

Match: Yes

8 Request Params

req.params is an object containing all parameters.

Example Output

```
{
  "id": "789",
  "userId": "456"
}
```

EXAMPLE CODE

```
1  const express = require('express');
2  const app = express();
3
4  // Get user by ID
5  app.get('/users/:id', (req, res) => {
6    res.send('User ID is ${req.params.id}');
7  });
8
9  // Get user posts
10 app.get('/users/:userId/posts/:postId', (req, res) => {
11   res.send('User ${req.params.userId} - Post ${req.params.postId}');
12 });
```

HOW IT WORKS

Express matches the URL pattern, captures values and makes them available in `req.params`.

QUICK REFERENCE TABLE

PARAMETER	EXAMPLE ROUTE	EXAMPLE URL	req.params OUTPUT
:id	/users/:id	/users/101	{ id: "101" }
:userId	/users/:userId	/users/55	{ userId: "55" }
:slug	/blog/:slug	/blog/node-js-guide	{ slug: "node-js-guide" }
Multiple	/users/:userId/posts/:postId	/users/7/posts/99	{ userId: "7", postId: "99" }

Pro Tip
Use meaningful names for parameters like :id, :userId, :slug for better readability.

Note
Route parameters capture only the path segments, not query strings.

Also Works With
Works with all HTTP methods: GET, POST, PUT, PATCH, DELETE

KEY TAKEAWAY
Route parameters make your routes dynamic and powerful by capturing values from the URL and using them in your application logic.

Express JS

EXPRESS.JS QUERY PARAMETERS

Query Parameters are used to filter, search, sort, and paginate data in a URL.

Read Data from the URL Query String using **req.query**

```
const express = require('express');
const app = express();

app.get('/users', (req, res) => {
  console.log(req.query);
  res.send('Query Parameters!');
});
```

1 req.query

Access query parameters using req.query object.

```
app.get('/search', (req, res) => {
  console.log(req.query);
});
```

Output: `req.query = { key: 'value' }`

2 Search

Use query params to search data.

Example URL: `/search?query=express`

`req.query.query → "express"`

3 Filter

Filter results by specific fields.

Example URL: `/products?category=books`

`req.query.category → "books"`

4 Sort

Sort results using query parameters.

Example URL: `/products?sort=price&order=asc`

```
req.query.sort  → "price"
req.query.order → "asc"
```

5 Pagination

Pagination helps in handling large data.

Example URL: `/products?page=2&limit=10`

```
req.query.page  → "2"
req.query.limit → "10"
```

6 Limit

Limit the number of results returned.

Example URL: `/users?limit=5`

`req.query.limit → "5"`

7 Page

Specify the page number.

Example URL: `/users?page=3`

`req.query.page → "3"`

8 URL Query

Everything after ? is the query string.

Example URL: `/users?name=jeet&age=25`

`?name=jeet&age=25`

EXAMPLE CODE

```
1  const express = require('express');
2  const app = express());
3
4  app.get('/users', (req, res) => {
5    const { search, category, sort, order, page, limit } = req.query;
6
7    res.json({
8      search: search || null,
9      category: category || null,
10     sort: sort || 'createdAt',
11     order: order || 'desc',
12     page: page ? parseInt(page) : 1,
13     limit: limit ? parseInt(limit) : 10
14   });
15 });
16
17 app.listen(3000, () => console.log('Server running on port 3000'));
```

HOW QUERY PARAMETERS WORK

Query parameters make your API flexible and powerful by allowing clients to control the response.

COMMON QUERY PARAMETER EXAMPLES

USE CASE	EXAMPLE URL	DESCRIPTION	req.query OUTPUT
Search	/search?query=nodejs	Search for nodejs	{ query: 'nodejs' }
Filter	/products?category=books	Filter by category	{ category: 'books' }
Sort	/products?sort=price&order=asc	Sort by price (ascending)	{ sort: 'price', order: 'asc' }
Pagination	/products?page=2&limit=10	Page 2 with 10 items	{ page: '2', limit: '10' }
Limit	/users?limit=5	Limit results to 5	{ limit: '5' }
Multiple	/users?name=jeet&age=25	Multiple parameters	{ name: 'jeet', age: '25' }

QUICK SYNTAX

Route: /api/data?key1=value1&key2=value2

Access in code: req.query.key1, req.query.key2

Pro Tip

Always validate and sanitize query data before using it in your application.

Note

Query parameters are visible in the URL. Do not use them for sensitive data.

Default Values

Provide default values for page, limit, sort, etc. for better user experience.

Combine Filters

You can combine multiple query parameters to make advanced and powerful queries.

KEY TAKEAWAY

Query parameters (req.query) help you build dynamic APIs for searching, filtering, sorting, and pagination easily.

Express

EXPRESS.JS REQUEST OBJECT

The request object (req) is passed to every route handler and middleware function.

The req object contains all information about the HTTP request.

```
const express = require('express');
const app = express();

app.get('/demo', (req, res) => {
 console.log(req);
 res.send('Request Object Example');
});
```

1 req.body

Contains data sent in the request body (POST/PUT).

Example

POST /users

```
{
  "name": "Jeet",
  "email": "jeet@example.com"
}
```

`req.body.name → "Jeet"`

2 req.params

Contains route parameters from the URL path.

Example Route

GET /users/:id

```
/users/123
```

`req.params.id → "123"`

3 req.query

Contains query string parameters from the URL.

Example URL

/search?name=jeet&page=2

```
?name=jeet&page=2
```

```
req.query.name → "jeet"
req.query.page → "2"
```

4 req.headers

Contains headers sent by the client.

Example Header

Content-Type: application/json

```
content-Type: application/json
user-agent: PostmanRuntime/7.36
accept: */*
```

`req.headers['content-type'] → "application/json"`

5 req.cookies

Contains cookies sent by the client.

Example Cookie

token=abc123; theme=dark

```
token = abc123
theme = dark
```

`req.cookies.token → "abc123"`

6 req.method

Contains the HTTP method used.

Example

POST /users

```
POST
```

`req.method → "POST"`

7 req.url

Contains the URL of the request.

Example

/users/123?active=true

```
/users/123?active=true
```

`req.url → "/users/123?active=true"`

8 req.ip

Contains the IP address of the client.

Example

```
192.168.1.25
```

`req.ip → "192.168.1.25"`

EXAMPLE: USING REQUEST OBJECT

```
1  const express = require('express');
2  const app = express();
3  app.use(express.json());
4
5  app.post('/users/:id', (req, res) => {
6    console.log('BODY:', req.body);
7    console.log('PARAMS:', req.params);
8    console.log('QUERY:', req.query);
9    console.log('HEADERS:', req.headers);
10   console.log('COOKIES:', req.cookies);
11   console.log('METHOD:', req.method);
12   console.log('URL:', req.url);
12   console.log('IP:', req.ip);
13   res.json({ message: 'Request received successfully!' });
14 });
15
16 app.listen(3000, () => console.log('Server running on port 3000'));
```

REQUEST FLOW

How Express Creates req Object

When a request comes in, Express parses the HTTP request and builds the req object with all the details. This object is then passed to your route handler or middleware.

QUICK REFERENCE TABLE

PROPERTY	TYPE	DESCRIPTION	EXAMPLE VALUE
req.body	Object	Data sent in request body	{ name: "Jeet" }
req.params	Object	Route parameters from URL	{ id: "123" }
req.query	Object	Query string parameters	{ name: "jeet", page: "2" }
req.headers	Object	Headers sent by client	{ "content-type": "application/json" }
req.cookies	Object	Cookies sent by client	{ token: "abc123" }
req.method	String	HTTP method of the request	"POST"
req.url	String	URL of the request	"/users/123?active=true"
req.ip	String	IP address of the client	"192.168.1.25"

Pro Tip

Always validate and sanitize data from req.body, req.query and req.params.

Security Note

Never trust user input. Always validate and use libraries like express-validator.

Important

Use express.json() or express.urlencoded() to parse req.body in your app.

Key Takeaway

The req object is your main source of information about the incoming request. Use it wisely!

Express

EXPRESS.JS RESPONSE OBJECT

Send Data Back to the Client

The response object **(res)** is used to send a response back to the client.

```
const express = require('express');
const app = express();

app.get('//', (req, res) => {
 res.send('Hello from Express!');
});
```

1 res.send()

Sends a response (text, HTML or data) to the client.

Example

```
res.send('Hello World!');
```

2 res.json()

Sends JSON data to the client.

Example

```
res.json({
  success: true,
  message: 'Data sent'
});
```

3 res.status()

404

Sets the HTTP status code.

Example

```
res.status(404)
  .send('Not Found');
```

4 res.redirect()

Redirects the client to another URL.

Example

```
res.redirect('/login');
```

5 res.download()

Forces download of a file.

Example

```
res.download(
  'files/report.pdf'
);
```

6 res.render()

Renders a view template (e.g. EJS, Pug).

Example

```
res.render('home', {
  title: 'My App'
});
```

7 res.end()

Ends the response without any data.

Example

```
res.end();
```

Complete Example

```
const express = require('express');
const app = express();

app.get('/api/user', (req, res) => {
  const user = { id: 1, name: 'Jeet', role: 'Developer' };

  if (user) {
    res.status(200).json(user);          // JSON response
  } else {
    res.status(404).sendd('User not found');
  }
});

app.listen(3000, () => console.log('Server running on port 3800'));
```

Quick Reference Table

Method	Purpose	Example
res.send()	Send text/HTML/data	res.send('Hello')
res.json()	Send JSON response	res.json({ ok: true })
res.status()	Set HTTP status code	res.status(201)
res.redirect()	Redirect to another URL	res.redirect('/login')
res.download()	Download a file	res.download('file.pdf')
res.render()	Render view template	res.render('index')
res.end()	End the response	res.end()

Response Flow

Using the response object, Express sends the final result back to the client in the form of text, JSON, file or HTML view.

Pro Tip
Use res.json() for API responses — it's clean, fast and easy to work with.

Common Mistake
Don't forget to set the correct status code using res.status() for better API handling.

Note
res.render() works with template engines like EJS, Pug, Handlebars, etc.

Key Takeaway
The response object (res) gives you full control to send, format and manage data for the client.

Express

EXPRESS.JS MIDDLEWARE BASICS

Middleware functions execute during the request-response cycle. They can modify `req`, `res`, end the request-response cycle, or call `next()` to pass control to the next middleware.

Middleware functions have access to the request object (`req`), the response object (`res`), and the `next()` function.

```
const express = require('express');
const app = express();

app.use((req, res, next) => {
  console.log('Middleware Executed');
  next(); // Pass control to next middleware
});

app.get('/', (req, res) => {
  res.send('Hello Express!');
});
```

1 Middleware

Functions that run during the request-response cycle.

Example: Custom logger, auth, parser, etc.

2 app.use()

Mounts middleware functions in the application.

Example: `app.use(myMiddleware);`

3 next()

Passes control to the next middleware function.

Example: `next();`

4 Request (req)

Contains information about the HTTP request.

Example: `req.method, req.url, req.body`

5 Response (res)

Used to send a response back to the client.

Example: `res.send(), res.json(), res.end()`

6 Logger

Logs request details like method, URL, time, etc.

Example: Console or file logger

7 Authentication

Verifies user identity before granting access.

Example: JWT, Sessions, API Keys

8 Pipeline

Middleware runs in order like a pipeline.

Example: `MW1 → MW2 → Route Handler`

MIDDLEWARE PIPELINE EXAMPLE

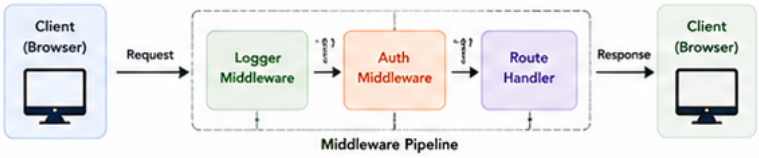
```
const express = require('express');
const app = express();

// Logger Middleware
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url} - ${new Date().toISOString()}`);
  next();
});

// Authentication Middleware
app.use((req, res, next) => {
  if (req.headers['authorization']) return next();
  res.status(401).send('Unauthorized');
});

// Route Handler
app.get('/', (req, res) => {
  res.send('Welcome to Express!');
});
```

REQUEST-RESPONSE CYCLE WITH MIDDLEWARE



1. **Logger Middleware** — Logs the request details
2. **Auth Middleware** — Verifies user identity
3. **Route Handler** — Handles the request and sends response

How it works:
Each middleware can perform tasks, modify `req/res`, end the cycle, or call `next()` to pass control.

COMMON BUILT-IN MIDDLEWARE

MIDDLEWARE	DESCRIPTION	USAGE
express.json()	Parses JSON request bodies	app.use(express.json())
express.urlencoded()	Parses URL-encoded bodies	app.use(express.urlenceded({ extended: true }))
express.static()	Serves static files	app.use(express.static('public'))
cookie-parser	Parses cookies	app.use(cookieParser())
morgan	HTTP request logger	app.use(morgan('dev'))
cors	Enables CORS	app.use(cors())
helmet	Sets security HTTP headers	app.use(helmet())

PRO TIP
Order matters! Middleware is executed in the order it is defined using `app.use()`.

BEST PRACTICE
Keep middleware small and focused. One task per middleware is ideal.

REMEMBER
Always call `next()` unless you are sending a response or ending the cycle.

AVOID
Forgetting `next()` will stop the request from reaching the next middleware or route.

KEY TAKEAWAY
Middleware makes your Express app modular, secure, and easier to maintain.

Express

EXPRESS.JS
TYPES OF MIDDLEWARE

Middleware functions have access to req, res, and next(). They can execute any code, change request/response objects, end the cycle, or call next() to pass control.

Middleware comes in different types. Each has a special role in the request-response cycle.

```
const express = require('express');
const app = express();

app.use((req, res, next) => {
   console.log('Middleware Executed');
   next();
});

app.listen(3000, () =>
  console.log('Server running on port 3000')
);
```

1 Application Middleware

Bound to an instance of app using app.use().

Example

```
app.use((req, res, next) => {
  console.log('App MW');
  next();
});
```

Use: Global tasks like logging, auth, parsing, etc.

2 Router Middleware

Bound to express.Router() instance.

Example

```
const router = express.Router();
router.use((req, res, next) => {
  console.log('Router MW');
  next();
});
```

Use: Specific to routes handled by the router.

3 Built-in Middleware

Built into Express. No need to install.

Example

- express.json()
- express.urlencoded()
- express.static()
- express.jsonp()

Use: Common tasks like parsing JSON, serving static files, etc.

4 Third Party Middleware

Created by the community. Install via npm.

Example

- morgan (logger)
- cors
- helmet
- cookie-parser

Use: Add powerful features easily to your app.

5 Error Handling Middleware

Handles errors in the application.

Example

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something
  went wrong!');
});
```

Use: Handle and respond to errors gracefully.

6 Custom Middleware

User-defined middleware for custom logic.

Example

```
app.use((req, req, next) => {
  // custom logic
  next();
});
```

Use: For custom checks, validation, or any logic.

7 Static Files Middleware

Serves static files like HTML, CSS, JS, images.

Example

```
app.use(express.static('public'));
// Files in /public are served
// as static assets
```

Use: Serve static assets like CSS, JS, images, etc.

8 JSON Parser Middleware

Parses incoming JSON request bodies.

Example

```
app.use(express.json());
// req.body will contain
// parsed JSON
```

Use: Parse JSON data in request body.

MIDDLEWARE EXECUTION ORDER (IMPORTANT!)

Middleware runs in the order it is defined.

If next() is not called, the request will stop at that middleware.
If a response is sent, Express will skip the remaining middleware.

COMMON BUILT-IN MIDDLEWARE

Middleware	Purpose	Usage
express.json()	Parse JSON bodies	app.use(express.json())
express.urlencoded()	Parse URL-encoded bodies	app.use(express.urlencoded({ extended: true }))
express.static()	Serve static files	app.use(express.static('public'))
express.jsonp()	Enable JSONP support	app.use(express.jsonp())

TIP
You can use multiple middleware in a row. Each one can do its job and pass control to the next middleware using next().

QUICK SUMMARY TABLE

Type	Bound With	Example Use Case
Application	app.use()	Logging, Auth, Global Config
Router	router.use()	Route group specific logic
Built-in	Express	Parsing, Static files, JSON
Third Party	npm package	Logging, CORS, Security, etc.
Error Handling	app.use(err, req, res, next)	Handle errors
Custom	app.use()	Custom checks, validation
Static Files	express.static()	Serve CSS, JS, Images
JSON Parser	express.json()	Parse JSON request body

PRO TIP
Keep middleware functions small and focused. One task per middleware is the best practice.

PERFORMANCE
Place heavy middleware (like loggers) after essential ones (like auth) for better performance.

BEST PRACTICE
Always call next() unless you are sending a response or ending the request.

SECURITY
Use security middleware like helmet, cors, and validator to protect your application.

KEY TAKEAWAY
Middleware makes your Express app modular, maintainable, and powerful!

Express JS

EXPRESS.JS BUILT-IN MIDDLEWARE

Built-in middleware functions come with Express.js and are ready to use. They help handle request data, serve static files, and more.

Express.js provides many **built-in middleware** functions for common tasks.

```
const express = require('express');
const app = express();

app.use(express.json());
app.use(express.urlencoded());
app.use(express.static('public'));

app.listen(3000, () => {
  console.log('Server running!');
});
```

1 express.json()

Parses incoming JSON data from the request body.

- Reads JSON data
- Converts to JavaScript object
- Useful for APIs

2 express.urlencoded()

Parses URL-encoded data from the request body.

- Reads form data
- Uses x-www-form-urlencoded
- Useful for HTML forms

3 express.static()

Serves static files like HTML, CSS, JS, images from a folder.

- Serves static assets
- From specified folder
- Improves app performance

4 Body Parser

Reads and parses data from the request body.

- Handles JSON and form data
- Simplifies request body access
- Adds data to req.body

5 Assets

Serves files like images, icons, and others.

- Serves public assets
- Used for frontend files
- Supports images, icons, etc.

Example Usage Flow

Quick Summary Table

Middleware	Purpose	Handles
express.json()	Parse JSON request body	JSON data (application/json)
express.urlencoded()	Parse form data	URL-encoded data
express.static()	Serve static files	HTML, CSS, JS, images
Body Parser	Parse request body	JSON and form data
Assets	Serve public assets	Images, icons, other files

Key Takeaways:

- Uses built-in middleware by default.
- Simplifies request data handling.
- Serves static files from public folder.
- Helps build secure and powerful Express apps.

Express

CUSTOM MIDDLEWARE

Write your own middleware functions to extend **Express** and handle requests in your own way! 🚀

Custom middleware are functions that you create to perform tasks during the request-response cycle.

```
const express = require('express');
const app = express();

// Use custom middleware
app.use(logger);
app.use(authenticate);

app.get('/', (req, res) => {
  res.send('Hello from Express!');
});

app.listen(3000, () => {
 console.log('Server running on port 3000');
});
```

CUSTOM MIDDLEWARE EXAMPLE

```
const express = require('express');
const app = express();

// 1. Logger Middleware
function logger(req, res, next) {
  console.log(`${req.method} ${req.url} - ${new Date().toISOString()}`);
  next(); // Pass control to the next middleware
})
// 2. Authentication Middleware
function authenticate(req, res, next) {
  const token = req.headers.authorization;
  if (!token) {
    return res.status(401).json({ error: 'Unauthorized' });
  }
  next();
}
// 3. Use Middleware
app.use(logger);          // Global Middleware
app.use(authenticate);    // Global Middleware

app.get('/', (req, res) => {
  res.send('Welcome! You are authenticated.');
});

app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

TIP
Always call `next()` to pass control to the next middleware.
If you don't, the request will hang!

COMMON USES OF CUSTOM MIDDLEWARE

LOGGER
Log details of requests.

```
console.log(
  `[${req.method}
  ${req.url}`
);
```

AUTHENTICATION
Verify user identity using tokens, API keys, etc.

```
if (!token)
 return res.status(401)
 .json({ error:
 'Unauthorized' });
```

VALIDATION
Validate request body, params or query.

```
if (!req.body.name)
 return res.status(400)
 .json({ error: 'Name
 is required' });
```

AUTHORIZATION
Check user roles and permissions.

```
if (!req.user.isAdmin)
 return res.status(403)
 .json({ error:
 'Forbidden' });
```

GLOBAL MIDDLEWARE
Applies to all routes and requests.

```
app.use(myMiddleware);
// Runs for every
 request
```

REUSABLE
Create once and use in multiple places.

```
app.use('/api',
  apiMiddleware);
app.use(otherMiddleware);
```

HOW CUSTOM MIDDLEWARE WORKS (THE FLOW)

Each middleware does its job and calls `next()` to pass control.
The last middleware or route handler sends the response.

SIMPLE CUSTOM MIDDLEWARE TEMPLATE

```
// Custom Middleware Template
function myMiddleware(req, res, next) {
  // Do something with req, res
  // ...

  // Pass control to next middleware
  next();
}

// Use it
app.use(myMiddleware);
// or for a specific route
app.get('/profile', myMiddleware, (req, res) => {
  res.send('Profile Page');
});
```

BENEFITS
- Keeps code clean & modular
- Reusable across routes
- Easy to test & maintain
- Great for scalable apps

BEST PRACTICE
- Keep middleware small & focused
- Do one thing at a time
- Always call next() or send response

AVOID
- Forgetting next()
- Too many middleware in one file
- Blocking the request

SECURITY TIP
- Always validate & sanitize input
- Never trust user data
- Use HTTPS and secure headers

KEY TAKEAWAY
Custom middleware gives you power to handle requests your way. Build, reuse and scale like a pro!

Express JS

EXPRESS ROUTER

Create modular, scalable & organized routes in your Express application. 🚀

Express Router is a built-in middleware that helps you group routes together and keep your code clean, modular and easy to maintain.

```
const express = require('express');
const app = express();

const userRouter = require('./routes/users');
app.use('/users', userRouter);

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

1 express.Router()

Creates a new router object to define routes.

```
const router = express.Router();
```

2 Routes

Define GET, POST, PUT, DELETE and more routes.

```
router.get('/', handler);
router.post('/', handler);
```

3 Controllers

Keep business logic in controllers.

```
router.get('/', userController.getAll);
```

4 Modules

Split your app into small, feature modules.

```
routes/, controllers/,
models/, middlewares/
```

5 Import

Import routers into your main app file.

```
const userRouter =
require('./routes/users');
```

6 Export

Export router to use in other files.

```
module.exports = router;
```

7 Organized Code

Clean structure, easy to read & maintain.

```
Better structure
Better productivity
```

8 Router File

Separate file for routes makes code cleaner.

```
users.js, auth.js,
product.js, etc.
```

PROJECT STRUCTURE EXAMPLE

my-express-app/	
app.js	Main application file
routes/	All route files
users.js	User related routes
auth.js	Auth related routes
products.js	Product related routes
controllers/	Business logic files
userController.js	User logic
authController.js	Auth logic
productController.js	Product logic
middlewares/	Custom middlewares
authMiddleware.js	Authentication middleware
models/	Database models

HOW EXPRESS ROUTER WORKS

1. Request comes to the server
2. Main app forwards to the router
3. Router matches the route and calls controller
4. Controller processes and sends response

Router helps you keep routes separate and your code clean, especially in large applications.

CODE EXAMPLE

1. ROUTER FILE
routes/users.js

```
const express = require('express');
const router = express.Router();

const userController =
  require('./controllers/userController');

router.get('/', userController.getAll);
router.get('/:id', userController.getById);
router.post('/', userController.create);

module.exports = router;
```

2. CONTROLLER FILE
controllers/userController.js

```
exports.getAll = (req, res) => {
  res.json({ message: 'All Users' });
};
exports.getById = (req, res) => {
  res.json({ id: req.params.id });
};
exports.create = (req, res) => {
  res.status(201).json({
    message: 'User Created'
  });
};
```

3. MAIN APP FILE
app.js

```
const express = require('express');
const app = express();

const userRouter =
  require('./routes/users');

app.use('/users', userRouter);

app.listen(3000, () => {
  console.log('Server running
    on port 3000');
});
```

PRO TIP
Use Router for feature-based structure. One feature = one router file.

BEST PRACTICE
Keep routes thin and move business logic to controllers.

CODE QUALITY
Modular code is easier to test, debug and maintain.

SCALABILITY
Router helps your app scale as it grows.

SECURITY
You can apply middleware per router or route.

KEY TAKEAWAY
Express Router = Clean Code, Better Structure, Happy Developer! 🙂

Express

REST API

Build scalable and secure APIs with Express.js 🚀

REST API (Representational State Transfer) is an architecture style for building web services using standard HTTP methods.

JSON — Data format used in API

Stateless — Each request is independent

Resource Based — Everything is a resource

Standard Methods — Use HTTP methods for operations

Status Codes — Standard HTTP responses

```
const express = require('express');
const app = express();
app.use(express.json());
app.get('/api/users', (req, res) => {
 res.json({ message: 'Users fetched successfully' });
});
app.listen(3000, () => {
 console.log('Server running on port 3000');
});
```

HTTP METHODS (CRUD OPERATIONS)

Method	Description	Example Endpoint	Example Use
GET	Retrieve data from the server	GET /api/users	Get all users
POST	Create new resource	POST /api/users	Create a new user
PUT	Update entire resource	PUT /api/users/1	Update user with ID 1
PATCH	Update part of resource	PATCH /api/users/1	Update user name with ID 1
DELETE	Delete resource from server	DELETE /api/users/1	Delete user with ID 1

JSON (DATA FORMAT)

APIs usually send and receive data in JSON format.

Example JSON Response

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john@example.com",
  "role": "Admin"
}
```

Example JSON Request (POST)

```
{
  "name": "Jane Doe",
  "email": "jane@example.com",
  "role": "User"
}
```

STATUS CODES

Code	Meaning	Use
200	OK	Request succeeded
201	Created	Resource created successfully
204	No Content	Request succeeded, no content to return
400	Bad Request	Invalid request or validation error
401	Unauthorized	Authentication required
403	Forbidden	No permission to access resource
404	Not Found	Resource not found
500	Internal Server Error	Something went wrong on server

ENDPOINTS EXAMPLE

Base URL: https://api.example.com

Endpoint		Description
GET	/api/users	Get all users
GET	/api/users/1	Get user by ID
POST	/api/users	Create new user
PUT	/api/users/1	Update user (full)
PATCH	/api/users/1	Update user (partial)
DELETE	/api/users/1	Delete user by ID

TIP
Use nouns for resources, not verbs.
Example: /api/users (good) not /api/getUsers (bad)

HOW REST API WORKS

Client (Send Request) → HTTP Method (GET, POST, PUT, DELETE) → Endpoint (/api/users) → Server (Express.js) → Response (JSON + Status Code) → Client Receives Response

EXAMPLE: GET /api/users/1

Request

```
GET /api/users/1 HTTP/1.1
Host: api.example.com
Authorization: Bearer token123
Accept: application/json
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 1,
  "name": "John Doe",
  "email": "john@example.com"
}
```

JSON

BEST PRACTICES

- Use proper HTTP methods
- Use meaningful endpoints
- Return proper status codes
- Keep API stateless

SECURITY TIPS

- Use HTTPS always
- Validate and sanitize input
- Use authentication (JWT/OAuth)
- Rate limit your APIs

DESIGN TIPS

- Version your API (/api/v1/...)
- Use consistent response format
- Handle errors gracefully
- Write clear documentation

TOOLS TO TEST

- Postman
- Insomnia
- Thunder Client (VS Code)
- curl

KEY TAKEAWAY

REST APIs make your application scalable, interoperable and easy to use across different platforms and devices.

Express

CRUD OPERATIONS

Create, Read, Update & Delete data using Express.js and Database

CRUD stands for Create, Read, Update and Delete. These are the 4 basic operations performed on data in any application.

```
const express = require('express');
const app = express();
app.use(express.json());

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

DATABASE

CREATE

Create new data

HTTP Method	POST
Endpoint	/api/users
Description	Create a new user

Example Request (JSON)

```
{
  "name": "John Doe",
  "email": "john@example.com",
  "age": 25
}
```

cURL Example

```
curl -X POST http://localhost:3000/api/users \
-H "Content-Type: application/json" \
-d '{"name":"John Doe","email":"john@example.com","age":25}'
```

READ

Read / Get data

HTTP Method	GET
Endpoint	/api/users/:id
Description	Get user by ID

Example Response (JSON)

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john@example.com",
} "age": 25
```

cURL Example

```
curl -X GET http://localhost:3000/api/users/1
```

UPDATE

Update existing data

HTTP Method	PUT
Endpoint	/api/users/:id
Description	Update user by ID

Example Request (JSON)

```
{
  "name": "John Smith",
  "email": "johnsmith@example.com",
  "age": 26
}
```

cURL Example

```
curl -X PUT http://localhost:3000/api/users/ \
-H "Content-Type: application/json" \
-d '{"name":"John Smith","email":"johnsmith@example.com","age":26}'
```

DELETE

Delete data

HTTP Method	DELETE
Endpoint	/api/users/:id
Description	Delete user by ID

Example Response (JSON)

```
{
  "message": "User deleted successfully",
  "id": 1
}
```

cURL Example

```
curl -X DELETE http://localhost:3000/api/users/1
```

REQUEST

Client sends a request to the server with required data (JSON, Params, Query, Headers, etc.)

RESPONSE

Server processes the request, performs operation and sends back an appropriate response with status code and data.

VALIDATION

Always validate and sanitize user input before saving or updating data in the database.

COMMON STATUS CODES

Code	Meaning	Use
200	OK	Request successful
201	Created	Resource created successfully
400	Bad Request	Invalid request or validation error
404	Not Found	Resource not found
500	Internal Server Error	Something went wrong on server

HOW CRUD OPERATIONS WORK

Client (Send Request) → Request (HTTP Method + Data) → Validation (Check & Validate Input) → Express.js Route (Handle Request) → Database (Perform Operation) → Response (Send Back Result) → Client (Show Result)

The server receives the request, validates data, performs the database operation and sends a response.

CRUD OPERATIONS SUMMARY

Operation	Method	Endpoint	Purpose	Example
Create	POST	/api/users	Create new user	Add a new user
Read	GET	/api/users/:id	Get user by ID	Get user with id 1
Update	PUT	/api/users/:id	Update user by ID	Update user with id 1
Delete	DELETE	/api/users/:id	Delete user by ID	Delete user with id 1

BEST PRACTICE

- Use proper HTTP methods
- Keep API endpoints RESTful
- Return meaningful responses

VALIDATION TIP

- Validate all inputs
- Sanitize user data
- Prevent injection attacks

SECURITY TIP

- Use authentication & authorization
- Don't expose sensitive data
- Use environment variables

DATABASE TIP

- Use proper indexing
- Handle errors properly
- Backup your database

PRO TIP

Well designed CRUD operations make your API powerful, secure and easy to maintain.

Express

HTTP STATUS CODES

HTTP Status Codes are standard responses from the server that tell the client the result of their request.

Understand the most common HTTP Status Codes used in REST APIs built with Express.js

```
app.get('/api/users', (req, res) => {
  res.status(200).json({
    success: true,
    message: 'Users fetched successfully'
  });
});
```

200 OK

The request was successful. The server returned the requested data.

Example Use

GET /api/users
(Get all users)

Example Response

```
{
  "success": true,
  "data": [ ... ]
}
```

201 Created

The request was successful and a new resource was created.

Example Use

POST /api/users
(Create a new user)

Example Response

```
{
  "success": true,
  "message": "User created",
  "id": 101
}
```

400 Bad Request

The server cannot process the request due to invalid syntax or parameters.

Example Use

Invalid data, missing required fields

Example Response

```
{
  "success": false,
  "message": "Invalid request",
  "errors": { ... }
}
```

401 Unauthorized

Authentication is required and has failed or not been provided.

Example Use

Accessing protected routes without token

Example Response

```
{
  "success": false,
  "message": "Unauthorized",
  "code": "NO_TOKEN"
}
```

403 Forbidden

The server understood the request but refuses to authorize it.

Example Use

User doesn't have permission to access the resource

Example Response

```
{
  "success": false,
  "message": "Unauthiized",
  "code": "NO_PERMISSION"
}
```

404 Not Found

The requested resource could not be found on the server.

Example Use

Resource or endpoint does not exist

Example Response

```
{
  "success": false,
  "message": "Not Found",
  "code": "NOT_FOUND"
}
```

500 Server Error

The server encountered an unexpected condition that prevented it from fulfilling the request.

Example Use

Unexpected server-side error

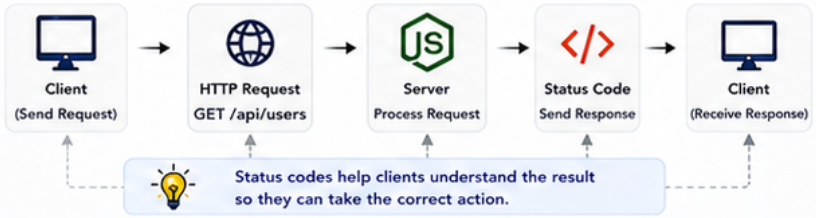
Example Response

```
{
  "success": false,
  "message": "Internal Server Error",
  "code": "SERVER_ERROR"
}
```

STATUS CODE CATEGORIES

- **1xx** Informational – Request received, continuing process
- **2xx** Success – The request was successfully received, understood, and accepted
- **3xx** Redirection – Further action needs to be taken to complete the request
- **4xx** Client Error – The request contains bad syntax or cannot be fulfilled
- **5xx** Server Error – The server failed to fulfill an apparently valid request

HOW IT WORKS (REQUEST → RESPONSE)



TIPS FOR DEVELOPERS

- Always return the correct status code.
- Use meaningful error messages.
- Validate input to avoid 400 errors.
- Use 401 for authentication issues.
- Use 403 for permission/authorization issues.
- Log 500 errors and fix server-side issues.

PRO TIP
Good use of status codes makes your API professional, reliable and easy to debug.

BEST PRACTICE
Follow REST standards and return consistent responses.

SECURITY TIP
Never expose sensitive information in error messages.

IMPROVES UX
Clear status codes help clients handle responses gracefully.

CLEAN & SCALABLE
Well-structured APIs are easier to maintain and scale.

Express.js

ERROR HANDLING

Handle errors gracefully in Express.js applications

Good error handling makes your API reliable, secure and easy to debug. Handle errors gracefully and send meaningful responses.

Prevent Crash | Better User Experience | Easy Debugging | Secure API | Reliable Application

```
app.get('/users/:id', async (req, res, next) => {
  try {
    const user = await getUserById(req.params.id);
    if (!user) return next(createError(404, 'User not found'));
    res.status(200).json(user);
  } catch (error) {
    next(error); // Pass error to error middleware
  }
});
```

TRY / CATCH

try }

Use try/catch to handle synchronous and asynchronous errors in your code.

```
try {
  const data = await doSomething();
  res.json(data);
} catch (error) {
  next(error); // Send error to
               // error middleware
}
```

NEXT(ERROR)

next (error)

Pass the error to Express using next(error) to handle it in middleware.

```
app.get('/', (req, res, next) => {
  try {
    // code that may fail
  } catch (error) {
    next(error); // forward error
  }
});
```

404 NOT FOUND

404

Used when the requested resource or endpoint cannot be found.

```
if (!user) {
  return next(createError(404,
  'User not found'));
}
```

Status: 404
Meaning: Resource not found

500 SERVER ERROR

500

Used when something goes wrong on the server while processing the request.

```
next(createError(500,
  'Something went wrong on
  server'));
```

Status: 500
Meaning: Internal Server Error

CUSTOM ERROR

Create custom error objects with status codes and custom messages.

```
class CustomError extends Error {
  constructor(statusCode, message) {
    super(message);
    this.statusCode = statusCode;
  }
}
```

ERROR MIDDLEWARE

Error middleware is a function with 4 arguments: (err, req, res, next)

- Catches all errors passed using next(error)
- Sends a proper response to the client
- Prevents server crash

```
// Error Handling Middleware (must be last)
app.use((err, req, res, next) => {
  const status = err.statusCode || 500;
  const message = err.message || 'Internal Server Error';

  // Log error (for debugging)
  connole.error('Error:  ${message}');

  res.status(status).json({
    success: false,
    status,
    message,
    stack: process.env.NODE_ENV === 'development' ? err.stack : undefined)
  })
});
```

EXAMPLE: FLOW

Client (Request) → Route / Controller (try/catch) → next(error) → Error Middleware → Response (JSON)

Example Response

```
{
  "success": false,
  "status": 404,
  "message": "User not found"
}
```

LOGGING

Log errors for tracking and debugging.

- Console Logging
- Winston
- Morgan
- Pino
- Bunyan

DEBUGGING TIPS

- Check error message & stack trace
- Use console.log() or debugger
- Test API using Postman / Thunder Client
- Check request body, params, query
- Validate database connection
- Use environment-based logs

Tip: Show stack trace only in development, not in production.

BEST PRACTICES

- Always use try/catch in async functions
- Use next(error) to pass errors
- Create custom error classes
- Keep error messages user-friendly

COMMON MISTAKES

- ✖ Not using try/catch in async code
- ✖ Forgetting to call next(error)
- ✖ Sending multiple responses
- ✖ Exposing stack trace in production

SECURITY TIP

Never expose sensitive information (database errors, tokens, stack trace) in production responses.

PERFORMANCE TIP

Avoid heavy operations in error middleware. Keep it simple and fast.

CUSTOM ERROR HELPER

```
const createError = (status, message) => {
  const err = new Error(message);
  err.statusCode = status;
  return err;
};
```

Express

ASYNC & AWAIT

Write asynchronous code that looks synchronous!

Async & Await is the modern way to write asynchronous code in JavaScript. It makes Promises easier to read, write and maintain.

```
const express = require('express');
const app = express();

app.get('/user/:id', async (req, res) => {
  try {
    const user = await getUserById(req.params.id);
    res.status(200).json({ success: true, data: user });
  } catch (error) {
    console.error('Error:', error.message);
    res.status(500).json({ success: false, message: 'Server Error' })
  }
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

ASYNC

`async`

The async keyword is used to declare an async function.

AWAIT

`await`

The await keyword is used to wait for a Promise to resolve.

PROMISE

A Promise represents the eventual completion (or failure) of an asynchronous operation.

TRY

try block lets you test a block of code for errors.

CATCH

catch block lets you handle errors that occur in try block.

API CALL

Use await with API calls to get data from external services.

DATABASE

Use await with database queries to perform CRUD operations.

ERROR HANDLING

Handle errors gracefully using try...catch and send proper responses.

ASYNC FLOW (HOW IT WORKS)

await pauses the function until the Promise settles (resolved or rejected), then continues.

EXAMPLE: API CALL WITH ASYNC/AWAIT

```
async function fetchUser(id) {
  try {
    const res = await fetch('https://api.example.com/users/${id}');
    if (!res.ok) throw new Error('Failed to fetch user');
    const data = await res.json();
    return data;
  } catch (error) {
    console.error('API Error:', error.message);
    throw error; // rethrow to handle in calling function
  }
}
```

await waits for the fetch() Promise to resolve and then processes the response.

EXAMPLE: DATABASE OPERATION

```
const { MongoClient } = require('mongodb');
const client = new MongoClient(process.env.MONGO_URL);

async function getUserById(id) {
  try {
    await client.connect();
    const db = client.db('mydb');
    const users = db.collection('users');
    const user = await users.findOne({ _id: id });
    return user;
  } catch (error) {
    console.error('DB Error:', error.message);
    throw error;
  } finally {
    await client.close();
  }
}
```

ERROR HANDLING PATTERN

```
app.get('/users', async (req, res, next) => {
  try {
    const users = await getAllUsers();
    res.status(200).json({ success: true, data: users });
  } catch (err) {
    next(err); // pass error to error middlewere
  }
});

// Error Middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(err.statusCode || 500).json({
    success: false,
    message: err.message || 'Something went wrong!'
  })
});
```

PROMISE vs ASYNC/AWAIT

Using Promises

```
getUserById(id)
  .then(user => {
    return getOrders(user.id);
  })
  .then(orders => {
    res.json(orders);
  })
  .catch(err => {
    res.status(500).json({ message: err.message });
  });
```

VS

Using Async/Await

```
async function getData(id) {
  try {
    const user = await getUserById(id);
    const orders = await getOrders(user.id);
    res.json(orders);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
}
```

COMMON USE CASES

- API Calls (REST, GraphQL)
- Database Operations
- File Upload & Processing
- Email / SMS Sending
- Payment Gateway
- Background Jobs

ASYNC/AWAIT BEST PRACTICES

- Always use try/catch to handle errors.
- Use meaningful error messages.
- Avoid blocking the event loop.
- Close DB connections in finally block.
- Keep async functions small and focused.
- Use Promise.all(:) for parallel operations.

QUICK EXAMPLE: PARALLEL CALLS

```
async function getDashboardData() {
  try {
    const [users, orders, stats] = await Promise.all([
      getAllUsers(),
      getAllOrders(),
      getStats()
    ]);
    return { users, orders, stats };
  } catch (err) {
    throw err;
  }
}
```

PRO TIP
Async/Await makes code cleaner, easier to debug and less error-prone.

DEBUGGING TIP
Use console.log(:) and try/catch to find and fix issues quickly.

ERROR TIP
Never ignore errors. Always handle Promise rejections.

SECURITY TIP
Do not expose sensitive info in error messages.

PERFORMANCE TIP
Use Promise.all(:) to run tasks in parallel and improve performance.

Express

ENVIRONMENT VARIABLES

Manage configuration and secrets securely in Express.js apps

Environment Variables store configuration outside your code. They keep secrets safe and make your app flexible across different environments.

.env File
Store Variables → Load with dotenv
at runtime → Access via
process.env → Use in App
(DB, API, etc.)

```
require('dotenv').config();

const port = process.env.PORT || 3000;
const dbUrl = process.env.DATABASE_URL;
const secret = process.env.SECRET_KEY;

app.listen(port, () => {
  console.log('Server running on port ${port}');
});
```

DOTENV

dotenv is a zero-dependency module that loads environment variables from a .env file into process.env.

Install

```
npm install dotenv
```

.ENV FILE EXAMPLE

.env

```
# Server Configuration
PORT=5000

# Database Configuration
DATABASE_URL=mongodb://localhost:27017/myapp

# Security
SECRET_KEY=my_super_secret_key_123
JWT_EXPIRES_IN=7d

# External APIs
API_KEY=abc123xyz789
MAIL_HOST=smtp.gmail.com
MAIL_USER=your_email@gmail.com
MAIL_PASS=your_email_password
```

IMPORTANT ENV VARIABLES (EXAMPLE)

Variable	Description	Example
PORT	Port number on which your server runs	5000
SECRET_KEY	Secret key for JWT or sessions	my_super_secret_key_123
DATABASE_URL	Database connection string	mongodb://localhost:27017/myapp
API_KEY	API key for external services	abc123xyz789
MAIL_USER	Email username for sending mails	your_email@gmail.com
MAIL_PASS	Email password or app password	your_email_password

USING ENV VARIABLES IN CODE

```
// Load environment variables
require('dotenv').config();

// Access variables
const port = process.env.PORT || 3000;
const dbUrl = process.env.DATABASE_URL;
const secretKey = process.env.SECRET_KEY;
const apiKey = process.env.API_KEY;

console.log('Port:', port);
console.log('DB:', dbUrl);

// Example usage
app.listen(port, () => {
   console.log( 'Server running on port ${port}');
});
```

PROCESS.ENV

node

process.env is a global object in Node.js that holds all environment variables.

```
console.log(process.env);
// returns all env variables
```

HOW IT WORKS

.env File (Key=Value) → dotenv.config() Loads Variables → process.env (Available Globally) → Your App (Use Anywhere)

.env file is loaded at runtime and variables are available in process.env

CONFIGURATION BY ENVIRONMENT

.env.development	.env.production	.env.test
PORT=5000	PORT=8080	PORT=5001
NODE_ENV=development	NODE_ENV=production	NODE_ENV=test
DATABASE_URL=...	DATABASE_URL=...	DATABASE_URL=...
DEBUG=true	DEBUG=false	DEBUG=true

Use different .env files for each environment.

WHY USE ENV VARIABLES?

- Keep secrets out of source code (GitHub, public repos)
- Easily change configuration without modifying code
- Manage different settings for development, testing and production
- Improve security and maintainability
- 12-Factor App best practice

BEST PRACTICES

- Never commit .env file to git
- Add .env to .gitignore
- Use strong secret keys
- Use environment-specific .env files

SECURITY TIP

- Store sensitive keys securely
- Use environment variables in hosting platforms
- Rotate keys regularly

DEBUGGING TIP

- Use console.log(process.env.VARIABLE) to check if variables are loaded.
- Make sure require('dotenv').config() is at the top of your entry file.

COMMON MISTAKES

- Forgetting to load dotenv
- Typos in variable names
- Using .env values without fallback
- Committing .env to repository

QUICK REFERENCE

```
require('dotenv').config();
```

- process.env.VARIABLE_NAME

Example: process.env.PORT

Express

STATIC FILES

Serve static files like CSS, JavaScript, Images and more using express.static() in Express.js

What are Static Files?

Static files are files that are served as they are to the client. They don't change unless modified.

CSS	JavaScript	Images	Assets	HTML	Downloads
Styles and design	Add interactivity	JPG, PNG, SVG, GIF	Fonts, Icons, Media, etc.	Web pages and templates	PDF, ZIP, Docs, etc.

```
const express = require('express');
const app = express();

// Serve static files from 'public' folder
app.use(express.static('public'));

app.listen(3000, () => {
console.log('Server running on http://localhost:3000');
});
```

Static files are available to the world via URL

PUBLIC FOLDER STRUCTURE (EXAMPLE)

```
my-express-app/
  public/
    css/
      style.css
    js/
      app.js
    images/
      logo.png
      hero.jpg
    assets/
      fonts/
      icons/
    downloads/
      file.pdf
    index.html
  server.js
```

public/

This folder is exposed to the internet. All files inside can be accessed by URL.

Best Practice

- Keep only public assets in public folder.
- Never put sensitive files inside.
- Organize files in sub-folders.

HOW express.static() WORKS

Client (Browser) → GET Request /css/style.css → Express Server express.static() → Public Folder (Reads File) → Response (Sends File)

Example URL Access

http://localhost:3000/css/style.css	→ Loads CSS file
http://localhost:3000/js/app.js	→ Loads JavaScript file
http://localhost:3000/images/logo.png	→ Loads Image file
http://localhost:3000/downloads/file.pdf	→ Downloads file
http://localhost:3000/index.html	→ Loads HTML file

CODE EXAMPLE

```
// server.js
const express = require('express');
const path = require('path');
const app = express();

// Serve static files from 'public' folder
app.use(express.static(path.join(__dirname, 'public')));

// Other routes
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

STATIC FILES FLOW (DETAILED)

1. **Browser Requests** — Browser requests a file like /css/style.css
2. **Express Checks** — Express checks if the file exists in the public folder
3. **File Found** — If found, Express sends the file as response
4. **File Delivered** — Browser receives the file (CSS, JS, Image, etc.)
5. **Page Rendered** — Browser uses the file to render the page properly

TIPS & NOTES

- express.static() serves the file based on the request path.
- The folder you provide is the root for static files.
- Files are served with correct Content-Type automatically.
- Great for CSS, JS, Images, Fonts, Videos, Downloads, etc.
- Keep your project clean and organized.

+ Common Use Cases

- Website assets (CSS, JS, Images)
- Admin dashboard files
- Downloads (PDF, Docs, ZIP)
- Icons, Fonts, Media files

Main Function	Key Method	Default Folder	Access via URL	Secure Your App
Serve static files to the client	express.static()	Any folder (e.g., public/)	http://localhost:3000/...	Keep sensitive files outside public folder

Express

TEMPLATE ENGINES

Build dynamic HTML pages using Template Engines in Express.js

Template engines let you separate your HTML from JavaScript logic and render dynamic content easily.

Views — Folder for template files (.ejs, .pug, etc.)

Layouts — Base layout for consistent design

Partials — Reusable pieces (header, footer, etc.)

Render — Render templates with data

Data Binding — Pass data and show dynamic content

```
const express = require('express');
const app = express();

// Set EJS as the view engine
app.set('view engine', 'ejs');
app.set('views', './views');

app.get('/', (req, res) => {
  const user = { name: 'CodeSardar', role: 'Developer'};
  res.render('home', { title: 'Dashboard', user });
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

Render dynamic HTML with ease!

PROJECT STRUCTURE (EXAMPLE)

```
my-express-app/
├── views/
│   ├── layouts/
│   │   └── layout.ejs
│   ├── partials/
│   │   ├── header.ejs
│   │   └── footer.ejs
│   ├── home.ejs
│   └── about.ejs
├── public/
│   ├── css/
│   └── js/
└── app.js
```

The views folder contains all template files that are rendered by Express.

POPULAR TEMPLATE ENGINES

EJS
Embedded JavaScript

- Simple & powerful
- Full JavaScript support
- Most popular with Express

Example (home.ejs)

```
<h1><%= title %></h1>
<p>Welcome, <%= user.name %>!</p>
<ul>
  <% users.forEach(u => { %>
    <li><%= u %></li>
  <% }) %>
</ul>
```

Pug
(formerly Jade)

- Clean & indentation based
- Less code, more readable
- Great for large projects

Example (home.pug)

```
h1= title
p Welcome, #{user.name}!
ul
  each u in users
    li= u
```

Handlebars
Logic-less templates

- Easy & beginner friendly
- No JavaScript in templates
- Great for simple apps

Example (home.hbs)

```
<h1>{{title}}</h1>
<p>Welcome, {{user.name}}!</p>
<ul>
  {{#each users}}
    <li>{{this}}</li>
  {{/each}}
</ul>
```

LAYOUTS & PARTIALS (EXAMPLE)

layouts/layout.ejs

```
<!DOCTYPE html>
<html>
 <head>
   <title><%= title %></title>
   <link rel="stylesheet" href="/css/style.css">
 </head>
 <body>
   <%- include('../partials/header') %>

   <main>
     <%- body %>
   </main>
   <%- include('../partials/footer') %>
   <script src="/s/app.js"></ccript>
 </body>
</ntml>
```

partials/header.ejs

```
<header>
  <nav>
    <a href="/">Home</a> |
    <a href="/about">About</a>
  </naw>
</header>
```

partials/footer.ejs

```
<footer>
  <p>&copy; 2024 CodeSardar</p>
</footer>
```

Use layouts to avoid repetition and partials for reusable components.

HOW RENDERING WORKS

Client Request (GET /) → Express Route (res.render) → Find View File in /views → Inject Data into Template → HTML Generated & Sent → Browser Displays HTML

res.render(view, data) combines the template with data and sends HTML to the client.

EXAMPLE: RENDER OUTPUT (home.ejs)

http://localhost:3000/

Dashboard

Welcome, CodeSardar!

- Alice
- Bob
- Charlie

Data Passed:

```
{
 "title": "Dashboard",
 "user": { "name": "CodeSardar" },
 "users": ["Alice", "Bob", "Charlie"]
}
```

CONFIGURE VIEW ENGINE

```
// Use EJS
app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));

// Use Pug
app.set('view engine', 'pug');
// Use Handlebars
app.engine('hbs', exphbs.engine({
  extname: '.hbs',
  defaultLayout: 'layout', // in views/layouts
  layoutsDir: 'views/layouts',
  partialsDir: 'views/partials'
}));
app.set('view engine', 'hbs');
```

QUICK COMMANDS

```
# Install EJS
npm install ejs

# Install Pug
npm install pug

# Install Handlebars
npm install express-handlebars

# Install nodemon (dev)
npm install -D nodemon
```

BEST PRACTICES

- Keep templates clean and simple.
- Use layouts and partials for reuse.
- Escape user input to prevent XSS.
- Keep logic in routes, not in views.

TIPS

- Use `<%- %>` in EJS for HTML output.
- Use `<%= %>` in EJS for escaped output.
- Use partials for header, footer, sidebar.
- Pass only required data to views.

COMMON USE CASES

- Dashboard pages
- Blog / CMS websites
- E-commerce sites
- Admin panels

ADVANTAGES

- Separation of concerns
- Maintainable code
- Reusable components
- Faster development

REMEMBER

Template Engines help you write clean, dynamic and maintainable web pages in Express.js!

Express

MONGODB INTEGRATION

Connect, Store and Manage Data with MongoDB in Express.js

MongoDB is a NoSQL database that stores data in flexible, JSON-like documents. Mongoose is an ODM that helps us work with MongoDB easily in Express.js apps.

MongoDB	Database	Collection	Document	Compass	Connection	NoSQL	CRUD
NoSQL database that stores data in documents.	Container for collections.	Group of documents.	A record in MongoDB (JSON-like).	GUI tool to view & manage MongoDB.	Connect your app to MongoDB using Mongoose.	Schema-less and highly scalable.	Create, Read, Update, Delete operations.

server.js (MongoDB Connection)

```
const express = require('express');
const mongoose = require('mongoose');
require('dotenv').config();

const app = express();
app.use(express.json());

// Connect to MongoDB
mongoose.connect(process.env.MONGO_URI)
  .then(() => console.log('✅ MongoDB Connected'))
  .catch((err) => console.error('❌ Connection Error:', err));

app.get('/', (req, res) => {
  res.send('MongoDB is Connected with Express! 🚀');
});
app.listen(process.env.PORT || 3000,
  () => console.log('Server running on port ${process.env.PORT || 3000}'));
```

mongoDB

NoSQL Database

1. CONNECTION SETUP

1. Install Mongoose

```
npm install mongoose dotenv
```

2. Create .env file

```
MONGO_URI=mongodb://127.0.0.1:27017/myapp
PORT=3000
```

3. Connect in your app

```
mongoose.connect(process.env.MONGO_URI)
```

4. That's it! You are connected 🚀

2. CRUD OPERATIONS EXAMPLE (MONGOOSE)

CREATE — Insert a document

```
const User =
require('./models/User');

const createUser = async () => {
  const user = new User({
    name: 'Alice',
    email: 'alice@example.com',
    age: 22
  });
  await user.save();
  console.log('User Created');
};
```

READ — Find documents

```
const getUsers = async () => {
  const users = await User.find();
  console.log(users);
};
```

UPDATE — Update a document

```
const updateUser = async (id) => {
  const user = await
  User.findByIdAndUpdate(
    id,
    { age: 23 },
    { new: true }
  );
};
console.log('User Updated', user);
};
```

DELETE — Delete a document

```
const deleteUser = async (id) => {
  await User.findByIdAndDelete(id);
  console.log('User Deleted');
};
```

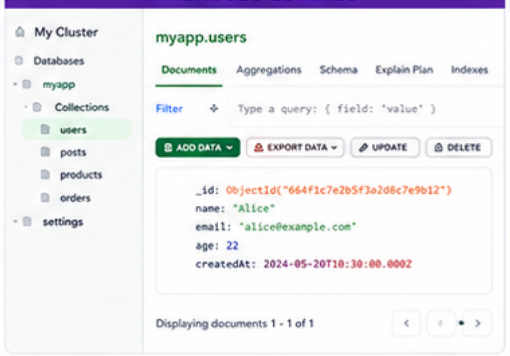
All operations are async and return Promises. Use await for better readability.

3. EXAMPLE DOCUMENT (IN USERS COLLECTION)

```
{
  "_id": "664f1c7e2b5f3a2d8c7e9b12",
  "name": "Alice",
  "email": "alice@example.com",
  "age": 22,
  "createdAt": "2024-05-20T10:30:00.000Z"
}
```

MongoDB Compass is the official GUI to manage databases, collections and documents easily.

4. MONGODB COMPASS



5. PROJECT STRUCTURE (EXAMPLE)

```
my-express-app/
├─ config/
│  └─ db.js          # Database connection
├─ models/
│  └─ User.js        # Mongoose schema
├─ routes/
│  └─ userRoutes.js
├─ .env              # Environment variables
├─ server.js         # Main server file
└─ package.json
```

6. MONGOOSE MODEL (USER.JS)

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number },
}, { timestamps: true });

module.exports = mongoose.model('User', userSchema);
```

Schema = Structure

Mongoose schema defines the structure of documents.

7. MONGODB (NoSQL) vs SQL

Feature	MongoDB (NoSQL)	SQL Database
Data Model	Document (JSON)	Table (Rows & Columns)
Schema	Schema-less	Fixed Schema
Relationships	Embedded Documents	Joins
Scalability	Horizontal Scaling	Vertical Scaling
Best For	Flexible, Large Scale Apps	Complex Transactions

8. COMMON CONNECTION STRINGS

Local MongoDB

```
mongodb://127.0.0.1:27017/myapp
```

Local with Auth

```
mongodb://username:password@127.0.0.1:27017/myapp
```

MongoDB Atlas (Cloud)

```
mongodb+srv://username:password@cluster0.xxxxx.mongodb.net/myapp?
retryWrites=true&w=majority
```

BEST PRACTICES

- Use Environment Variables for URI
- Handle Connection Errors
- Use async/await
- Close Connection on Exit

TIP

- Use .env to keep secrets safe
- Use Mongoose for better structure
- Validate data before saving

COMMON COMMANDS (Compass / Shell)

```
# show dbs                  # List databases
# use myapp                 # Switch database
# show collections          # List collections
# db.users.find()           # Find all documents
```

USE CASES

- User Management
- Blog / CMS
- E-commerce
- Real-time Apps

REMEMBER

MongoDB + Express.js = Powerful Combo for modern, scalable applications! 🚀

Express

MONGOOSE

Elegant MongoDB object modeling for Node.js

Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in validation, query building, middleware, and more.

Connection — Easy connection to MongoDB
Schema — Define structure and rules
Model — Compile schema into a Model
Document — Instances of Models
Validation — Built-in & custom validation
Populate — Reference docs made easy
Queries — Powerful & flexible query API
NoSQL — Built for MongoDB (NoSQL)

```
const mongoose = require('mongoose');
// Connect to MongoDB
mongoose.connect('mongodb://127.0.0.1:27017/myapp')
  .then(() => console.log('✅ MongoDB Connected'))
  .catch(err => console.error('❌ Connection Error:', err));

// Connection Events
mongoose.connection.on('connected', () => console.log('Mongoose Connected'));
mongoose.connection.on('error', err => console.log('Mongoose Error:', err));
mongoose.connection.on('disconnected', () => console.log('Mongoose Disconnected'));
```

mongoose

MongoDB ODM for Node.js

1. CONNECTION

```
const mongoose = require('mongoose');

// Basic Connection
mongoose.connect('mongodb://127.0.0.1:27017/myapp')
  .then(() => console.log('✅ Connected'))
  .catch(err => console.error('❌ Error:', err));

// Connection Options
mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});
```

Useful Connection Events

- connected : Connected to DB
- error : Connection error
- disconnected : Disconnected from DB
- reconnected : Reconnected to DB

2. SCHEMA

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true, trim: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, min: 0 },
  role: { type: String, enum: ['user', 'admin'],
  default: 'user' },
  createdAt: { type: Date, default: Date.now }
}, { timestamps: true });
```

Schema defines the structure, data types, validation rules and default values.

3. MODEL

```
const User = mongoose.model('User', userSchema);
```

Schema → Compile → Model → User

Model is a constructor compiled from Schema.
We use Model to create, read, update and delete documents.

4. DOCUMENT

```
const user = new User({
  name: 'Alice',
  email: 'alice@example.com',
  age: 22,
  role: 'user'
});
await user.save();
// Document in MongoDB
{
  "_id": ObjectId("664f1c7e2b5f3a2d8c7e9b12"),
  "name": "Alice",
  "email": "alice@example.com",
  "age": 22,
  "role": "user",
  "createdAt": "2024-05-20T10:30:00.000Z",
  "updatedAt": "2024-05-20T10:30:00.000Z"
}
```

5. VALIDATION

```
const productSchema = new mongoose.Schema({
  title: {
    type: String,
    required: [true, 'Title is required'],
    minlength: [3, 'Title too short']
  },
  price: {
    type: Number,
    min: [0, 'Price cannot be negative'],
    required: true
  },
  email: {
    type: String,
    match: [/^[^\s@]+@[^\s@]+\.[^\s@]+$/', 'Invalid email']
  }
});
```

Mongoose provides built-in & custom validation.

6. POPULATE (REFERENCES)

User Collection: _id, name, email — ref: 'User' → Post Collection: _id, title, user: ObjectId

```
const postSchema = new mongoose.Schema({
  title: String,
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
});

// Populate User in Post
const posts = await Post.find().populate('user');
```

7. QUERIES

Find All	`User.find()`
Find One	`User.findOne({ email: 'a@a.com' })`
Filter	`User.find({ age: { $gt: 18 } })`
Select	`User.find().select('name email')`
Sort	`User.find().sort({ age: -1 })`
Limit	`User.find().limit(10)`
Skip	`User.find().skip(10)`
Count	`User.countDocuments({ role: 'user' })`
Update	`User.updateOne({ _id: id}, { $set: { name: 'Bob' }})`
Delete	`User.deleteOne({ _id: id })`

Mongoose provides a powerful query builder.
Supports chaining, conditions, operators & more.

8. CRUD OPERATIONS EXAMPLE

CREATE
```
const user = new User({ name: 'Alice', email: 'a@a.com' });
await user.save();
```

READ
```
const users = await User.find({});
const user = await User.findById(id);
```

UPDATE
```
await User.updateOne({ _id: id }, { $set: { name: 'Alice' }});
```

DELETE
```
await User.deleteOne({ _id: id });
await User.deleteMany({ role: 'user' });
```

9. MIDDLEWARE (HOOKS)

Pre Save Hook
```
userSchema.pre('save', function(next) {
  console.log('About to save user...');
  next();
});
```

Post Save Hook
```
userSchema.post('save', function(doc) {
  console.log('User saved:', doc.name);
});
```

Middleware lets you run functions before or after certain operations.

WHY USE MONGOOSE?

- Schema-based modeling
- Built-in validation
- Easy relationships with populate
- Powerful query API
- Middleware (hooks)
- Better code organization
- Less boilerplate code
- Works perfectly with Express.js

Mongoose makes working with MongoDB in Express.js simple, structured and efficient.

QUICK REFERENCE

Connect	`mongoose.connect(uri)`
Schema	`new mongoose.Schema()`
Model	`mongoose.model()`
Save	`doc.save()`
Find	`Model.find()`
Update	`Model.updateOne()`
Delete	`Model.deleteOne()`
Populate	`.populate('field')`

Tip: Use async/await for clean and readable code!

Express.js

AUTHENTICATION

Secure your Node.js apps with User Authentication

Authentication is the process of verifying who a user is.

We'll build a secure system with Register, Login, Password Hashing using bcrypt and protect routes.

User	Register	Login	Password	bcrypt	Hashing	Security	Authentication
Store user information securely	Create new user account	Verify user credentials	Never store plain passwords	Hash & salt passwords	Strong one-way encryption	Protect users & app data	Verify identity before access

```
const express = require('express');
const app = express();
const bcrypt = require('bcrypt');
const User = require('./models/User');

app.use(express.json());

// Example: Hash Password
const password = 'MySecurePass@123';
const salt = bcrypt.genSaltSync(10);
const hash = bcrypt.hashSync(password, salt);
console.log(hash); // Store this in DB, not plain password
```

1. USER MODEL (MONGOOSE)

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
}, { timestamps: true });

module.exports = mongoose.model('User', userSchema);
```

Always hash the password before saving!

2. REGISTER (SIGN UP)

```
const bcrypt = require('bcrypt');
const User = require('./models/User');

app.post('/register', async (req, res) => {
  try {
    const { name, email, password } = req.body;
    const exist = await User.findOne({ email });
    if (exist) return res.status(400).json({ message: 'Email already in use' });

    const salt = await bcrypt.genSalt(10);
    const hash = await bcrypt.hash(password, salt);

    const user = await User.create({ name, email, password: hash });
    res.status(201).json({ message: 'User registered successfully', user });

  } catch (err) {
    res.status(500).json({ message: 'Server error' });
});
```

3. LOGIN

```
app.post('/login', async (req, res) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ email });
    if (!user) return res.status(400).json({ message: 'Invalid credentials' });

    const match = await bcrypt.compare(password, user.password);
    if (!match) return res.status(400).json({ message: 'Invalid credentials' });

    // Create token (JWT example)
    const token = require('jsonwektoken').sign(
      { id: user._id, email: user.email },
      process.env.JWT_SECRET,
      { expiresIn: '7d' }
    );
    res.json({ message: 'Login successful', token });
  } catch (err) {
});  res.status(500).json({ message: 'Server error' });}
```

4. PASSWORD HASHING WITH BCRYPT

- bcrypt automatically salts the password.
- Same password = different hash.
- Slow by design to resist brute force attacks.

```
const bcrypt = require('bcrypt');

// Hash a password
const salt = await bcrypt.genSalt(10);
const hash = await bcrypt.hash('MyPass@123', salt);

// Compare a password
const isMatch = await bcrypt.compare('MyPass@123', hash);
console.log(isMatch); // true / false
```

5. PROTECT ROUTES (AUTH MIDDLEWARE)

```
const jwt = require('jsonwektoken');

function protect(req, res, next) {
  const token = req.haeders.authorization?.split(' ')[1];
  if (!token) return res.status(401).json({ message: 'Not authorized' });
  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    res.status(401).json({ message: 'Invalid token' });
}}
module.exports = protect;
```

6. USE PROTECTED ROUTE

```
const protect = require('./middleware/auth');

app.get('/profile', protect, async (req, res) => {
  const user = await User.findById(req.user.id).select('-password');
  res.json({ user });
});
```

Only logged-in users with a valid token can access this route.

7. ENVIRONMENT VARIABLES

```
PORT=5000
MONGO_URI=mongodb://127.0.0.1:27017/authdb
JWT_SECRET=your_super_secret_key_here
JWT_EXPIRES_IN=7d
```

Never commit .env file to GitHub!
Add .env to .gitignore

8. AUTH FLOW DIAGRAM

Token sent with each request in Header
Authorization: Bearer <token>

BEST PRACTICES

- Use strong passwords
- Hash passwords with bcrypt
- Use HTTPS in production
- Store secrets in .env
- Validate & sanitize inputs

SECURITY TIPS

- Never store plain passwords
- Use rate limiting on login
- Use JWT & short expiry
- Logout by blacklisting tokens
- Keep dependencies updated

COMMON MISTAKES

- Storing plain passwords
- Not validating user input
- Exposing JWT secret
- Sending token in Response body (instead use HttpOnly cookies in production)

TOOLS & PACKAGES

bcrypt	– Password hashing
jsonwoktoben	– JWT auth
mongoose	– MongoDB ODM
express-validator	– Input validation
dotenv	– Environment variables

QUICK COMMANDS

```
npm install express mongoose bcrypt jsonwektoben dotenv
npm install express-validator ---save
node server.js
# Start your auth server 🚀
```

Express

JWT AUTHENTICATION

Secure your APIs with JSON Web Tokens in Express.js

What is JWT?

JWT (JSON Web Token) is a compact and self-contained way to securely transmit information between parties as a JSON object.

- Stateless authentication
- Cross-domain friendly
- Widely supported
- Compact and lightweight

JWT	Token	Bearer	Secret Key	Sign	Verify	Authorization	Protected Routes
JSON Web Token Standard for secure information exchange	Compact, URL-safe token format	Sent in Authorization header as Bearer <token>	Used to sign and verify tokens	Create token using secret key	Validate token and its signature	Check user permissions and roles	Restrict access to secured endpoints

Example: Generate & Verify Token

```
const jwt = require('jsomoktoken');
const SECRET_KEY = 'my_super_secret_key';

// Sign (Generate) Token
const payload = { id: 1, email: 'user@example.cem', role: 'user' };
const token = jwt.sign(payload, SECRET_KEY, { expiresIn: '1h' });
console.log(token);

// Verify Token
try {
  const decoded = jwt.verify(token, SECRET_KEY);
  console.log('Token is valid:'; decoded);
} catch (err) {
  console.log('Invalid token:', err.message);
}
```

1. USER LOGIN (GET TOKEN)

```
app.post('/login', async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });

  if (!user) return res.status(401).json({ message: 'User not found' });

  const isMatch = await bcrypt.compare(password, user.password);
  if (!isMatch) return res.status(400).json({ message: 'Invalid password' });

  // Create JWT Token
  const token = jwt.sign(
    { id: user._id, email: user.email, role: user.role },
    process.env.JWT_SECRET,
    { expiresIn: '1h' }
  );
 res.json({ token, user: { id: user._id, email: user.email, role: user.role } });
});
```

On successful login, send the token to the client.

2. TOKEN STRUCTURE (PAYLOAD)

Header . Payload . Signature

Base64Url . Base64Url . Base64Url

```
{
  "id": "663fa2b1e4c8b24f8c9a1234",
  "email": "user@example.com",
  "role": "user",
  "iat": 1715683200,
  "exp": 1715686800,
}
```

- iat → Issued At (timestamp)
- exp → Expiration Time (timestamp)
- Do not store sensitive data in payload!

3. SEND TOKEN IN REQUEST

Use the token in every protected request.

Authorization Header

```
Authorization: Bearer <your_token_here>
```

Example with fetch

```
fetch('/api/profile', {
 headers: {
   'Authorization': 'Bearer ' + token
 }
});
```

Always use HTTPS in production!

4. VERIFY TOKEN (MIDDLEWARE)

```
const jwt = require('jsomoktoken');

const authenticate = (req, res, next) => {
 const authHaader = req.headers.authorization;

 if (!authHeader || !authHeader.startsWith('Bearer '))
  return res.status(401).json({ message: 'No token, authorization denied' });

 const token = authHaeder.split(' ')[1];

 jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
  if (err) return res.status(403).json({ message: 'Invalid or expired token' });

  req.user = user; // Attach decoded payload to request
  next();
 });
};

module.exports = authenticate;
```

5. PROTECTED ROUTE

```
const authenticate = require('./middleware/authenticate');

// Only authenticated users can access this
app.get('/api/profile', authenticate, async (req, res) => {
 try {
   const user = await User.findById(req.user.id).select('-password');
   res.json({ user });
 }
 } catch (error) {
   res.status(500).json({ message: 'Server error' });
 }
});
```

If token is valid, user info is available in req.user and request proceeds.

6. TOKEN EXPIRATION

Set token expiry while signing.

```
jwt.sign(payload, SECRET_KEY, { expiresIn: '1h' });
```

Common values: 15m, 1h, 7d, 30d

After expiration, user must login again.

7. LOGOUT (CLIENT SIDE)

JWT is stateless, so logout is handled on client.

- Delete token from localStorage / cookies
- Remove user data from state
- Token will no longer be sent

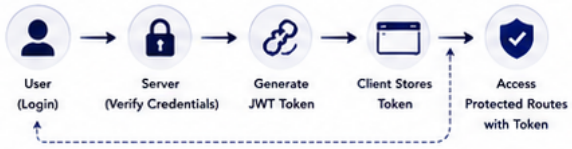
For advanced logout, use token blacklist with Redis or database.

8. ENVIRONMENT VARIABLES

```
JWT_SECRET=my_super_secret_key_change_this
JWT_EXPIRES_IN=1h
PORT=5000
MONGO_URI=mongodb://127.0.0.1:27017/myapp
```

Never hardcode secret key in your code!

9. COMPLETE AUTH FLOW



Server verifies token on every request using secret key.

10. JWT BEST PRACTICES

- Use a strong secret key
- Set appropriate expiration time
- Use HTTPS always
- Store token securely (HttpOnly cookie or secure storage)
- Do not store sensitive data in payload
- Implement role-based access control (RBAC)
- Rotate secret keys periodically

QUICK REFERENCE

- Sign: jwt.sign(payload, secret, options)
- Verify: jwt.verify(token, secret)
- Header: Authorication: Bearer <token>
- Status Codes: 401 (Unauthorized), 403 (Forbidden)

COMMON STATUS CODES

- 200 OK — Success
- 401 Unauthorized — No token / Invalid credentials
- 403 Forbidden — Invalid / Expired token
- 500 Server Error — Something went wrong

TOOLS

- jwt.io — Decode & verify tokens
- Postman — Test APIs
- Insomnia — API testing
- MongoDB — Store users

USE CASES

- User Authentication
- API Security
- Role Based Access
- Single Sign-On (SSO)

REMEMBER

JWT makes your API stateless, scalable and secure!
Keep your secret key safe and your users safer.

Express

COOKIES

Store small data in browser with Express.js

Cookies

A cookie is a small piece of text data stored in the browser. It helps remember user information between requests.

- **cookie-parser** — Middleware to parse cookies
- **Set Cookie** — Send cookie from server
- **Read Cookie** — Get cookie in request
- **HTTP Only** — Not accessible to JS (safer)
- **Secure** — Send only over HTTPS
- **SameSite** — Control cross-site requests
- **Expires** — Set expiry time (absolute)

Example: Set and Read Cookie

```
const express = require('express');
const cookieParser = require('cookie-parser');

const app = express();
app.use(cookieParser());

// Set a cookie
app.get('/set', (req, res) => {
   res.cookie('username', 'Alice', {
     httpOnly: true, secure: true, sameSite: 'lax',
     maxAge: 60 * 60 * 1000 // 1 hour
   });
    res.send('Cookie set!');
});
// Read a cookie
app.get('/read', (req, res) => {
   const user = req.cookies.username;
   res.send(`Welcome ${user || 'Guest'}`);
});
```

Cookies

Keep track of your session!

1. Set Cookie

```
res.cookie(name, value, options);

// Example:
res.cookie('token', 'xyz123', {
  httpOnly: true,
  secure: true,
  maxAge: 1000 * 60 * 60, // 1 hour
  path: '/',
  sameSite: 'lax'
});
```

Sends a cookie to the browser in Set-Cookie header.

2. Read Cookie

```
const cookieParser = require('cookie-parser');

app.use(cookieParser());

app.get('/read', (req, res) => {
  const token = req.cookies.token;
  res.send(`Token: ${token}`);
});
```

Reads cookies from req.cookies object (parsed).

3. HTTP Only Cookie

```
Set-Cookie: sessionId=abc123;
  HttpOnly; Secure; SameSite=Lax;
Expires=Wed, 21 May 2026 07:28:00 GMT
```

Not accessible by JavaScript

Helps prevent XSS attacks.

4. Cookie Options

httpOnly	→	Not accessible to JS
secure	→	Send only over HTTPS
sameSite	→	Lax / Strict / None
expires	→	Set expiry date/time
path	→	Cookie available on paths
domain	→	Share with subdomains

5. Signature + Security

- Use HTTP Only for sensitive data
- Use Secure in production
- Use SameSite to prevent CSRF
- Validate cookie values on server

Cookies + Good Options = Safer User Sessions!

Typical Cookie Flow

Keeps user logged in using cookies!

Common Use Cases

- User authentication (session)
- Remember user preferences
- Shopping cart (e-commerce)
- Theme / language settings
- Flash messages & notifications

Types Example

Session Cookie	→	Deleted on browser close
Persistent Cookie	→	Remains for set expiry time
First-party	→	Set by your own domain
Third-party	→	Set by another domain (e.g., analytics)

Quick Tips

- Always set HTTP Only for auth cookies
- Use Secure in live/production (HTTPS)
- Choose SameSite=Lax (best for most apps)
- Set reasonable Expires / maxAge
- Keep cookie size small (4KB limit)

Summary

- Simple way to store small data
- Automatic with cookie-parser
- Supports secure & safer options
- Great for session management
- Use with care for sensitive data

Better security. Better experience. Keep your users logged in securely with Cookies!

Express

SESSIONS

Manage user sessions on the server with express-session

Sessions store user information on the server and a Session ID in the browser cookie. It is perfect for login systems and user authentication.

express-session	Session ID	Login Session	Logout	Memory Store	Redis Store	Authentication	Store
Session middleware for Express.js	Unique ID stored in cookie	Create session after successful login	Destroy session on logout	Default store (small apps only)	Production ready with Redis	Keep user logged in securely	Where session data is stored

```
const express = require('express');
const session = require('express-session');
const app = express();

app.use(session({
  secret: 'CodeSardarSecretKey',  // Secret key
  resave: false,
  saveUninitialized: false,
  cookie: { httpOnly: true, secure: false, sameSite: 'lax', maxAge: 1000*60*60 },
  store: new session.MemoryStore() // Use RedisStore in production
}));

app.get('//', (req, res) => {
  res.send('Sessions are working! User: ' + (req.session.user || 'Guest'));
});
app.listen(5000, () => console.log('Server running on http://localhost:5000'));
```

Browser stores only Session ID in a cookie

1. INSTALL & SETUP

Install express-session

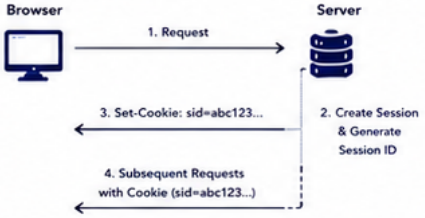
```
npm install express-session
```

Basic Setup

```
const session = require('express-session');
app.use(session({
  secret: 'your_secret_key',
  resave: false,
  saveUninitialized: false,
  cookie: { httpOnly: true, sameSite: 'lax' },
  store: new session.MemoryStore()
}));
```

Always use a strong secret key.

2. HOW SESSIONS WORK



Session data is stored on the server, not in the cookie.

3. LOGIN SESSION (EXAMPLE)

```
// POST /login
app.post('/login', (req, res) => {
  const { email, password } = req.body;
  const user = findUser(email, password); // your logic
  if (user) {
    req.session.user = {
      id: user.id,
      name: user.name,
      email: user.email
    };
    req.session.isLoggedIn = true;
    res.redirect('/dashboard');
  } else {
    res.send('Invalid credentials');
  }
});
```

Store only minimal data in session.

4. READ SESSION DATA

```
app.get('/dashboard', (req, res) => {
  if (!req.session.isLoggedIn) {
    return res.redirect('/login');
  }
  res.send(`Welcome ${req.session.user.name}`);
});
```

Common Session Properties	
req.session	Session object
req.session.user	User data
req.session.isLoggedIn	Login status
req.sessionID	Session ID

5. LOGOUT (DESTROY SESSION)

```
app.get('/dashboard', (req, res) => {
  req.session.destroy(err => {
    if (err) return res.send('Logout failed');
    res.clearCookie('connect.sid'); // Clear cookie
    res.redirect('/login');
  });
});
```

Always destroy session and clear cookie on logout.

6. SESSION ID & COOKIE

- The session ID (sid) is stored in a cookie named connect.sid by default.
- This ID is used to fetch session data from the store.

Default Cookie Options		
httpOnly	true	Not accessible via JavaScript
secure	false	Send only over HTTPS (set true in production)
sameSite	'lax'	CSRF protection
maxAge	1 hour	Example expiration time

7. SESSION STORES

Memory Store (Default)

- Stores session in server memory
- Easy to use
- Data lost on server restart
- Not for production

Redis Store (Recommended)

- Stores session in Redis
- Persistent & scalable
- Works across multiple servers
- Best for production

```
const RedisStore = require('connect-redis').default;
const { createClient } = require('redis');
// See Redis example in your code
```

8. AUTHENTICATION WITH SESSIONS

Check authentication middleware example:

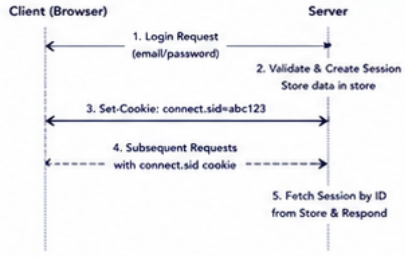
```
function requireLogin(req, res, next) {
  if (req.session.isLoggedIn) return next();
  res.redirect('/login');
}
// Use: app.get('/dashboard', requireLogin, (req,res)=>{ ... });
```

9. MEMORY STORE vs REDIS STORE

Feature	Memory Store	Redis Store
Data Location	Server Memory	Redis Database
Persistence	No (lost on restart)	Yes
Scalability	Single Server	Multiple Servers
Performance	Good (small apps)	Excellent
Production Ready	No	Yes

Use Redis Store in production for reliability and scalability.

10. SESSION FLOW DIAGRAM



BEST PRACTICES

- Use strong secret key
- Set httpOnly and secure in production
- Use Redis Store in production
- Destroy session on logout
- Store only necessary data

SECURITY TIPS

- Use HTTPS (secure: true)
- Use sameSite to prevent CSRF
- Regenerate session ID after login
- Set proper session expiration

COMMON SESSION METHODS

```
req.session                 // Get session object
req.session.destroy()       // Destroy session
req.session.regenerate(cb)  // Regenerate session ID
req.session.save(cb)        // Save session
req.sessionID               // Get session ID
```

TOOLS & PACKAGES

```
express-session   – Session middleware
connect-redis     – Redis store for sessions
redis             – Redis client
cookie-parser     – Parse cookies
```

IMPORTANT

Sessions are stored on the server, not in the client.
Keep your secret key secret and your users safe! 🔒

Express

FILE UPLOAD

Handle file uploads in Express.js using Multer

Multer is a Node.js middleware for handling multipart/form-data, mainly used for uploading files.

Multer	Upload	Storage	Disk Storage	Memory Storage	Single File	Multiple Files	File Filter
Middleware for file upload	Send files from client to server	Where files are stored	Save files on server disk	Store files in server memory	Upload one file	Upload multiple files	Accept/Reject files by type

```
const express = require('express');
const multer = require('multer');
const app = express();

app.use(express.json());

// Multer setup (Disk Storage Example)
const storage = multer.diskStorage({
  destination: (req, file, cb) => cb(null, 'uploads/'),
  filename: (req, file, cb) => {
    const uniqueName = Date.now() + '-' + file.originalname;
    cb(null, uniqueName);
  }
});
const upload = multer({
  storage: storage,
  fileFilter: (req, file, cb) => {
    const allowed = /jpeg|jpg|png|pdf/;
    const extname = allowed.test(path.extname(file.originalname).toLowerCase());
    const mimetype = allowed.test(file.mimetype);
    if (mimetype && extname) return cb(null, true);
    cb(new Error('Only images or PDF files are allowed!'));
  },
limits: { fileSize: 5 * 1024 * 1024 } // 5MB
```

Upload files easily and securely!

1. INSTALL MULTER

```
$ npm install multer
```

Multer is a middleware. It is not built-in in Express.

Works with multipart/form-data

2. BASIC SETUP

```
const express = require('express');
const multer = require('multer');
const app = express();

const upload = multer(); // memoryStorage (default)
app.use(express.json());

app.post('/upload', (req, res) => {
  res.send('Multer is working!');
});
```

Default storage is MemoryStorage.

3. STORAGE OPTIONS

Disk Storage
(Saves files on server)

Good for large files, persistent storage.

Memory Storage
(Saves files in memory)

Fast, temporary. Files not saved after server restart.

4. DISK STORAGE EXAMPLE

```
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/');
  },
  filename: (req, file, cb) => {
     const uniqueName = Date.now() + '-' + file.originalname;
     cb(null, uniqueName);
  }
});
const upload = multer({ storage: storage });
```

Files will be saved in /uploads folder.

5. MEMORY STORAGE EXAMPLE

```
const storage = multer.memoryStorage();
const upload = multer({ storage: storage });
```

Files are available in req.file.buffer
Useful for processing before saving.

6. INSTALE FILE UPLOAD

Field name in form: avatar

```
// HTML Form
<form action="/upload-avatar" method="POST"
   enctype="multipart/form-data">
   <input type="file" name="avatar" />
   <button type="submit">Upload</button>
</form>

// Route
app.post('/upload-avatar', upload.single('avatar'),
  (req, res) => {
  // req.file contains file info
  res.json({ message: 'File uploaded', file: req.file });

});
```

req.file	File information

7. MULTIPLE FILES UPLOAD

Field name in form: photos

```
// HTML Form
<form action="/upload-photos" method="POST"
   enctype="multipart/form-data">
   <input type="file" name="photos" multiple />
   <button type="submit">Upload</button>
</form>

// Route
app.post('/upload-photos', upload.array('photos', 5),
  (req, res) => {
  // req.files is an array
  res.json({ message: 'Files uploaded', files: req.files });

});
```

req.files[]	Array of files

8. FILE FILTER

```
const upload = multer({
  storage: storage,
  fileFilter: (req, file, cb) => {
    const allowed = /jpeg|jpg|png|pdf/;
    const extname = allowed.test(
     path.extname(file.originalname).toLowerCase()
  );
    const mimetype = allowed.test(file.mimetype);
    if (mimetype && extname) return cb(null, true);
    cb(new Error('Only images or PDF files are allowed!'));
  }
});
```

Filter by file extension & MIME type.

9. LIMITS (FILE SIZE)

```
const upload = multer({
  storage: storage,
  limits: { fileSize: 5 * 1024 * 1024 } // 5MB
});
```

Exceeding limit will throw error:
LIMIT_FILE_SIZE

10. AVAILABLE FILE PROPERTIES (req.file)

fieldname	→	Field name in form
originalname	→	Original file name
encoding	→	File encoding
mimetype	→	File MIME type
size	→	File size in bytes
filename	→	Saved file name (Disk Storage)
path	→	Full path (Disk Storage)
buffer	→	File buffer (Memory Storage)

BEST PRACTICES

- Always validate file type.
- Limit file size.
- Use strong filenames (avoid user input).
- Store files outside public directory if not needed.

SECURITY TIPS

- Never trust file extensions.
- Scan files for malware (if required).
- Use HTTPS for uploads.
- Set proper permissions on upload folder.

COMMON ERRORS

- LIMIT_FILE_SIZE – File too large
- LIMIT_UNEXPECTED_FILE – Wrong field name
- Unexpected field – Check form field name
- Only images or PDF files are allowed! – Filter error

TOOLS & PACKAGES

multer	–	File upload middleware
express	–	Web framework
fs	–	File system access
path	–	Path utilities

SUPPORTED FILE TYPES (Example)

Images: jpg, jpeg, png, gif, webp
Documents: pdf
Others: txt, csv, zip (as per your filter)

VALIDATION

Validate & sanitize user input in Express.js using express-validator

Why Validate?

- Prevent invalid data
- Improve security
- Return clear errors
- Better user experience

express-validator	Required	Email	Password	Min Length	Sanitize	Validation	Error Messages
Middleware for validating & sanitizing data	Ensure field is not empty	Validate email format	Validate password strength	Set minimum length for fields	Clean & escape user input	Check & collect validation errors	Show clear & user-friendly error messages

Basic Validation Flow

```
const { body, validationResult } = require('express-validator');

// Example: Register Validation
const registerValidation = [
  body('name').notEmpty().withMessage('Name is required'),
    .isLength({ min: 2 }).withMessage('Name must be at least 2 characters'),
  body('email').isEmail().withMessage('Please enter a valid email'),
  body('password').isLength({ min: 6 })
    .withMessage('Password must be at least 6 characters')
    .matches(/^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d@$!%*?&]{6,}$/)
    .withMessage('Password must include letters and numbers')
];
// Check for errors
const checkValidation = (req, res, next) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({
      success: false,
      errors: errors.array()
    });
  }
  next();
```

How it Works

Client Sends Request → Validation Rules Run → If Valid → Request Continues / If Invalid → Return Errors

1. REQUIRED

Ensure a field is not empty.

Example

```
body('username')
  .notEmpty().withMessage('Username is required')
```

Also Useful

- `.exists()` – Check if field exists
- `.not().isEmpty()` – Field should not be empty

2. EMAIL

Validate email format.

Example

```
body('email')
  .isEmail().withMessage('Enter a valid email')
  .normalizeEmail() // Convert to lowercase
```

Options

- `.isEmail()` – Valid email check
- `.normalizeEmail()` – Normalize email

3. PASSWORD

Validate password strength.

Example

```
body('password')
  .isLength({ min: 6 }).withMessage('Min 6 chars')
  .matches(/^(?=.*[A-Za-z])(?=.*\d).+$/)
  .withMessage('Use letters and numbers')
```

Options

- `.isLength({ min: 6 })`
- `.matches(regex)` – Custom regex rules

4. MIN LENGTH

Set minimum length for a field.

Example

```
body('name')
  .isLength({ min: 3 })
  .withMessage('Name must be at least 3 chars')
```

Options

- `.isLength({ min: n })`
- `.isLength({ max: n })` – Max length

5. SANITIZE

Clean and sanitize user input.

Examples

```
body('username').trim().escape();
body('age').toInt();
body('email').normalizeEmail();
```

Common Sanitizers

- `.trim()` – Remove spaces
- `.escape()` – Escape HTML
- `.toInt()` / `.toFloat()` – Convert type
- `.normalizeEmail()` – Normalize email

6. VALIDATION (CHECK)

Check validation and handle errors.

Example

```
const errors = validationResult(req);
if (!errors.isEmpty()) {
  return res.status(400).json({
    errors: errors.array()
  });
}
```

Result Format

```
[
  {
    "msg": "Email is required",
    "param": "email",
    "location": "body"
  }
]
```

7. ERROR MESSAGES

Return clear & user-friendly messages.

Example Response (400)

```
{
  "success": false,
  "errors": [
    {
      "msg": "Email is required",
      "param": "email",
      "location": "body"
    },
    {
      "msg": "Password must be at least
      "param": "password",
      "location": "body"
    }
  ]
},
```

8. PRACTICAL EXAMPLE: REGISTER ROUTE

```
const { body } = require('express-validator');
const registerValidation = [
  body('name').notEmpty().withMessage('Name is required')
    .isLength({ min: 2 }),
  body('email').isEmail().withMessage('Valid email required')
    .normalizeEmail();
  body('password').isLength({ min: 6 }).withMessage('Min 6 chars')
    .matches(/^(?=.*[A-Za-z])(?=.*\d).+$/)
    .withMessage('Use letters and numbers')
];
```

Use in Route

```
const express = require('express');
const { registerValidation, checkValidation } = require('./validation');
const router = express.Router();

router.post('/register',
  registerValidation,
  checkValidation,
  (req, res) => {
    // If valid, continue
    res.json({ success: true, message: 'Registered successfully!' });
  }
);
```

Controller (if valid)

```
const register = async (req, res) => {
  const { name, email, password } = req.body;

  // Save user to database
  res.json({ success: true, message: 'User
  registered!' });
};
```

Only valid data reaches your controller!

9. COMMON VALIDATIONS QUICK REFERENCE

Validation Rule	Description	Example
notEmpty()	Field should not be empty	body('name').notEmpty()
isEmail()	Check valid email	body('email').isEmail()
isLength({min})	Minimum length	body('pass').isLength({ min: 6 })
isLength({max})	Maximum length	body('bio').isLength({ max: 150 })
isInt() / toInt()	Check/Convert to integer	body('age').isInt().toInt()
isMobilePhone()	Validate mobile number	body('phone').isMobilePhone()
isURL()	Validate URL	body('website').isURL()
matches(regex)	Custom regex validation	body('pass').matches(/regex/)

10. BEST PRACTICES

- Always validate on server side.
- Keep rules simple and clear.
- Sanitize all user inputs.
- Return meaningful error messages.
- Use custom error messages.
- Validate only required fields.
- Use validation middlewares in separate file.

COMMON ERROR RESPONSES

- 400 Bad Request – Validation failed
- 422 Unprocessable Entity – Invalid data
- 500 Server Error – Something went wrong

TOOLS & PACKAGES

- express-validator – Validation & Sanitization
- helmet – Secure headers
- cors – Enable CORS
- morgan – HTTP request logger

TIP

Always validate and sanitize user input to build secure and reliable applications.

RESOURCES

- https://express-validator.github.io/docs/
- https://expressjs.com/
- https://nodejs.org/

KEY TAKEAWAY

Validate early, sanitize always, and return clear error messages for a better and secure API.

Express.js

Security is not a feature, it's a foundation. Build secure and reliable APIs.

SECURITY

Best practices to secure your Express.js application

```
// Security middleware in Express.js
const express = require('express');
const helmet = require('helmet');
const cors = require('cors');
const rateLimit = require('express-rate-limit');
const xss = require('xss-clean');
const mongoSanitize = require('express-mongo-sanitize');
const app = express();

// Apply Security Middlewares
app.use(helmet());
app.use(cors());
app.use(rateLimit({ windowMs: 15 * 60 * 1000, max: 100 }));
app.use(xss());
app.use(mongoSanitize());
app.listen(5000, () => console.log('✅ Secure API running on port 5000'));
```

SECURE API

- Protect
- Validate
- Sanitize
- Limit
- Secure

Build. Secure. Scale.

1 HELMET

Helps secure Express apps by setting various HTTP headers.

Install & Use

```
npm install helmet
```

```
app.use(helmet());
```

Benefits

- Set secure HTTP headers
- Prevent common attacks like XSS, clickjacking, etc.

2 CORS

Enable Cross-Origin Resource Sharing to control who can access your API.

Install & Use

```
npm install cors
```

```
app.use(cors({
  origin: 'https://yourdomain.com',
    methods: ['GET', 'POST'],
  credentials: true
}));
```

Benefits

- Allow trusted domains
- Block unauthorized domains

3 RATE LIMITER

Limit the number of requests from a single IP to prevent brute force & abuse.

Install & Use

```
npm install express-rate-limit
```

```
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 min
  max: 100,  // limit each IP
});
app.use(limiter);
```

Benefits

- Prevent brute force attacks
- Reduce server load & spam

4 XSS (CROSS SITE SCRIPTING)

Prevent injection of malicious scripts in user input.

Use xss-clean

```
npm install xss-clean
```

```
app.use(xss());
```

Benefits

- Clean user input
- Prevent XSS attacks

5 CSRF (CROSS SITE REQUEST FORGERY)

Protect against unauthorized actions performed on behalf of authenticated users.

Use csurf

```
npm install csurf
```

```
const csrf = require('csurf');
app.use(csrf({ cookie: true }));

// In forms or requests, send
// the CSRF token
```

Benefits

- Protect user sessions
- Prevent CSRF attacks

6 HTTPS

Always use HTTPS to encrypt data between client & server.

How to Use

- Get SSL certificate
- Enable HTTPS on server
- Redirect HTTP to HTTPS

```
// Redirect HTTP to HTTPS
app.use((req, res, next) => {
  if (req.secure ||
      req.headers['x-forwarded-proto']
      === 'https') return next();
  res.redirect('https://' + req.host
      + req.url);
});
```

Benefits

- Encrypt data in transit
- Prevent man-in-the-middle attacks

7 SANITIZE INPUT

Sanitize user input to prevent NoSQL injection and data manipulation.

Use express-mongo-sanitize

```
npm install express-mongo-sanitize
```

```
app.use(mongoSanitize());
```

Benefits

- Remove $ operators
- Prevent NoSQL injection

8 VALIDATE & ESCAPE INPUT

Validate and escape user input before processing.

Use express-validator

```
npm install express-validator
const { body, validationResult } =
  require('express-validator');
app.post('/api/user', [
  body('email').isEmail(),
  body('password').isLength({ min: 6 })
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors:
        errors.array() });
  }
  // proceed...
});
```

Benefits • Ensure valid data • Prevent invalid/malicious input

9 SECURE HEADERS

Set important security headers manually or via Helmet.

Example Headers

- Content-Security-Policy
- X-Content-Type-Options: nosniff
- X-Frame-Options: DENY
- Referrer-Policy: no-referrer
- Strict-Transport-Security

Benefits

- Control browser behavior
- Reduce attack surface

10 SECURE COOKIES

Use secure cookie settings to protect session cookies.

Example

```
res.cookie('token', token, {
  httpOnly: true,
  secure: true,     // true in https
  sameSite: 'Strict',
  maxAge: 1000 * 60 * 60
});
```

Benefits

- Prevent XSS from stealing cookies
- Reduce CSRF risk

11 LOG & MONITOR

Log suspicious activities and monitor your API.

Tips

- Log login attempts
- Monitor errors
- Use tools like Winston, Morgan
- Set up alerts

Benefits

- Detect and respond early
- Improve overall security

12 KEEP DEPENDENCIES UPDATED

Keep all dependencies up to date.

Commands

```
npm outdated
```

```
npm update
```

Benefits

- Fix known vulnerabilities
- Keep your app secure

COMMON THREATS

- XSS – Cross Site Scripting
- CSRF – Cross Site Request Forgery
- SQL Injection – Database Attacks
- NoSQL Injection – Query Manipulation
- Brute Force – Login Attacks
- DDoS – Denial of Service

SECURITY BEST PRACTICES

- Always use HTTPS
- Validate & sanitize all inputs
- Use least privilege access
- Rate limit your APIs
- Keep secrets in ENV variables
- Regularly update dependencies
- Use Helmet and other security middleware
- Backup your data regularly

USEFUL TOOLS & PACKAGES

- helmet – Secure HTTP headers
- cors – Enable CORS
- express-rate-limit – Rate limiting
- xss-clean – Prevent XSS
- express-mongo-sanitize – Prevent NoSQL injection
- csurf – CSRF protection
- express-validator – Input validation
- hpp – Prevent HTTP Parameter Pollution

SECURE API CHECKLIST

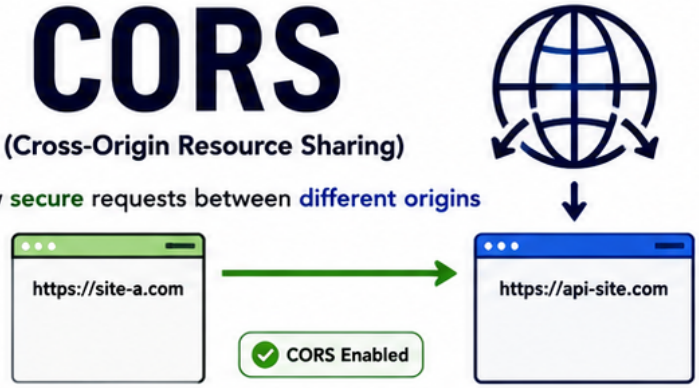
- HTTPS enabled
- Helmet configured
- CORS restricted
- Rate limiting enabled
- Input validated & sanitized
- XSS & CSRF protection
- Secure cookies
- Dependencies updated

Express JS

CORS

(Cross-Origin Resource Sharing)

Allow secure requests between different origins



WHAT IS CORS?

CORS is a security feature implemented by browsers to control how web pages in one origin can request resources from another origin.

ENABLE CORS IN EXPRESS.JS

```
const express = require('express');
const cors = require('cors');
const app = express();

// Enable CORS for all routes and origins (not recommended for production)
app.use(cors());

// Custom CORS configuration (recommended)
app.use(cors({
   origin: 'https://frontend.com',
   methods: ['GET', 'POST', 'PUT', 'PATCH', 'DELETE'],
   allowedHeaders: ['Content-Type', 'Authorization'],
   credentials: true
}));
app.listen(5000, () => console.log('Server running on port 5000'));
```

DEFAULT PORTS

HTTP : 80
HTTPS : 443

SAME ORIGIN POLICY

Browsers block requests across different origins for security. CORS allows servers to relax this restriction safely.

1. CORS

CORS lets a server specify any origin (domain) that is allowed to access its resources.

Key Point

CORS is not a server-side feature, it is enforced by the browser.

2. ORIGIN

Origin = Protocol + Domain + Port

Example:

```
https://example.com          (default 443)
http://example.com           (default 80)
https://example.com:3000     (different port)
```

Different origin = Different protocol, domain or port.

3. HEADERS

Headers allowed in the request.

Common Headers

- Content-Type
- Authorization
- X-Requested-With
- Accept

Use `allowedHeaders` to specify which headers are allowed.

4. METHODS

HTTP methods allowed for cross-origin requests.

Common Methods

GET POST PUT PATCH DELETE OPTIONS

Use `methods` to specify allowed HTTP methods.

5. CREDENTIALS

Allow cookies or authentication information to be sent with requests.

Option

```
credentials: true
```

- Requires `Access-Control-Allow-Credentials: true`
- Cannot use wildcard (*) for origin
- Use specific origin

6. ALLOW ORIGIN

Specifies which origins are allowed to access the resources.

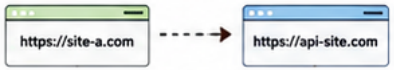
Examples

```
origin: 'https://frontend.com'      // Single origin
origin: ['https://a.com', 'https://b.com']   // Multiple
origin: '*'                         // All origins (Not secure)
```

Avoid using '*' in production when credentials are true.

7. CROSS ORIGIN

A request made from one origin to another origin.



Cross-Origin Request

Browser sends OPTIONS preflight request for non-simple requests.

8. SECURITY

CORS helps in maintaining security by controlling who can access your API.

Best Practices

- Allow only trusted origins
- Do not use '*' with credentials
- Restrict allowed headers and methods
- Always use HTTPS in production

9. PRELIGHT (OPTIONS)

Browser sends an OPTIONS request to check if the actual request is safe.

Preflight Request Includes

Origin
Access-Control-Request-Method
Access-Control-Request-Headers

Server responds with allowed rules.

10. RESPONSE HEADERS

Server must send these headers in response.

```
Access-Control-Allow-Origin
Access-Control-Allow-Methods
Access-Control-Allow-Headers
Access-Control-Allow-Credentials
Access-Control-Max-Age
```

CORS FLOW (HIGH LEVEL)

1. Browser → 2. Preflight (OPTIONS) → 3. Server Check → 4. Response with CORS Headers → 5. Actual Request → 6. Final Response

SAMPLE RESPONSE HEADERS

```
Access-Control-Allow-Origin: https://frontend.com
Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS
Access-Control-Allow-Headers: Content-Type, Authorization
Access-Control-Allow-Credentials: true
Access-Control-Max-Age: 86400
```

COMMON CORS ERRORS

- No 'Access-Control-Allow-Origin' header
- Crediental is not supported if the CORS header 'Access-Control-Allow-Origin' is '*'
- Request header field Authorization is not allowed by Access-Control-Allow-Headers
- Response to preflight request doesn't pass access control check

QUICK CHECKLIST

- ✓ Set correct origin
- ✓ Allow required methods
- ✓ Allow required headers
- ✓ Set credentials if needed
- ✓ Use HTTPS always

Express.js

HELMET

Secure your Express.js app by setting important HTTP headers automatically.

Helmet is a middleware that helps you secure your Express.js application by setting various HTTP headers.

Secure Headers | Protect App | Block Attacks | Improve Security | Best Practice

How to Use Helmet

```
const express = require('express');
const helmet = require('helmet');
const app = express();

// Use Helmet with default settings
app.use(helmet());
// Or customize specific features
 app.use(helmet({
   contentSecurityPolicy: false,
   hsts: { maxAge: 31536000, includeSubDomains: true }
}));

app.get('/', (req, res) => {
   res.send('✅ Secure Express App with Helmet!');
});
app.listen(5000, () => console.log('Server running on port 5000'));
```

WHAT DOES HELMET DO?

- Sets secure HTTP headers
- Helps mitigate common web vulnerabilities
- Protects against XSS, clickjacking, MIME sniffing and more
- Easy to use and highly customizable
- Best practice for every Express.js app

HELMET SECURITY HEADERS

Helmet sets 11 important security headers by default.

- Better Security
- Protect Users
- Prevent Attacks
- Follow Best Practices

1 CONTENT SECURITY POLICY (CSP)

Helps prevent XSS attacks by controlling resources the browser is allowed to load.

Header: `Content-Security-Policy`

Default: `default-src 'self'`

Prevents loading of unauthorized resources.

2 STRICT TRANSPORT SECURITY (HSTS)

Forces the browser to use HTTPS instead of HTTP.

Header: `Strict-Transport-Security`

Default: `max-age=31536000; includeSubDomains`

Protects against protocol downgrade attacks.

3 X-XSS-PROTECTION

Enables the browser's built-in XSS filtering.

Header: `X-XSS-Protection`

Default: `1; mode=block`

Helps stop reflected XSS attacks.

4 X-CONTENT-TYPE-OPTIONS

Prevents MIME type sniffing (MIME Sniffing Attack).

Header: `X-Content-Type-Options`

Default: `nosniff`

Prevents browsers from guessing content type.

5 X-FRAME-OPTIONS (FRAME GUARD)

Protects your site from clickjacking attacks.

Header: `X-Frame-Options`

Default: `SAMEORIGIN`

Prevents your site from being embedded in iframes.

6 REFERRER-POLICY

Controls how much referrer information is sent with requests.

Header: `Referrer-Policy`

Default: `no-referrer`

Protects user privacy by limiting referrer data.

7 PERMISSIONS-POLICY

Controls browser features & APIs that sites can use.

Header: `Permissions-Policy`

Default: `varies (restrictive by default)`

Disables powerful features not needed by your site.

8 CACHE-CONTROL

Helps prevent caching of sensitive responses.

Header: `Cache-Control`

Default: `no-cache`

Reduces risk of sensitive data being cached.

9 PRAGMA

Backwards compatibility for HTTP/1.0 caches.

Header: `Pragma`

Default: `no-cache`

Ensures old caches do not store sensitive data.

10 EXPECT-CT

Enables Expect-CT for Certificate Transparency.

Header: `Expect-CT`

Default: `max-age=86400, enforce`

Helps detect and prevent misissued SSL certificates.

11 CROSS-ORIGIN-OPENER-POLICY

Isolates your site in its own browsing context group.

Header: `Cross-Origin-Opener-Policy`

Default: `same-origin`

Protects against some cross-origin attacks & data leaks.

EXAMPLE RESPONSE HEADERS

```
content-security-policy: default-src 'self'
strict-transport-security: max-age=31536000; includeSubDomains
x-content-type-options: nosniff
x-xss-protection: 1; mode=block
x-frame-options: SAMEORIGIN
referrer-policy: no-referrer
permissions-policy: accelerometer=(), camera=()
cache-control: no-cache
pragma: no-cache
expect-ct: max-age=86400, enforce
cross-origin-opener-policy: same-origin
```

COMMON THREATS HELMET HELPS MITIGATE

- XSS (Cross Site Scripting)
- Clickjacking
- MIME Sniffing
- Man-in-the-Middle Attacks
- Data Leakage
- Insecure Transport

BEST PRACTICES

- Always use Helmet in every Express app
- Keep Helmet updated to the latest version
- Review and customize headers as per your needs
- Use HTTPS in production along with HSTS

CUSTOMIZE HELMET (Example)

```
app.use(helmet({
   contentSecurityPolicy: false,
   hsts: { maxAge: 31556000, includeSubDomains: true },
   frameguard: { action: 'deny' },
   referrerPolicy: { policy: 'strict-origin-when-cross-origin' }
}));
```

BEST SECURITY = TRUST + PROTECTION

Helmet makes your Express.js app more secure by default.

Secure today, safe tomorrow!

Express JS

LOGGING

in Express.js with Morgan

Better logs. Better insights. Better apps.

Morgan	Console	Request Logs	Error Logs	Monitoring	Performance	Analytics
HTTP request logger middleware	Logs printed in terminal	Track incoming requests	Capture & log errors	Monitor app health	Track response time & status	Analyze traffic & usage

HOW TO USE MORGAN

```
const express = require('express');
const morgan = require('morgan');
const app = express();

// Use morgan middleware
app.use(morgan('dev')); // 'combined', 'tiny', 'dev', 'common', 'short'

app.get('/', (req, res) => {
  res.send('Logging with Morgan');
});

app.listen(5000, () => {
  console.log('Server running on http://localhost:5000');
});
```

WHY LOGGING?

- Understand user behavior
- Detect & fix errors
- Monitor application health
- Improve performance
- Audit & security tracking
- Make data-driven decisions

1 MORGAN FORMATS

Format	Description
dev	Colored output for development
combined	Apache combined log format
common	Apache common log format
tiny	Minimal output
short	Shortened output
custom	Custom log format

Use 'dev' for development and 'combined' for production.

2 CONSOLE OUTPUT (dev)

```
GET / 200 12.345 ms - 150
POST /api/login 200 45.123 ms - 320
GET /api/users 404 3.214 ms - 22
PUT /api/users/1 200 22.456 ms - 180
DELETE /api/users/1 500 6.789 ms - 56
```

Method Status Response Time - Response Size

- GET, POST, PUT, DELETE...
- 200, 404, 500...

3 REQUEST LOGS (Example)

- Track every incoming request
- Log method, URL, status, response time
- Helps in finding slow endpoints
- Useful for traffic analysis

Example:

```
GET /api/products 200 33.21 ms - 512
POST /api/orders 201 120.10 ms - 256
GET /api/users 404 2.15 ms - 22
```

4 ERROR LOGS

- Capture errors and exceptions
- Log stack traces & error messages
- Helps in quick issue resolution
- Use with custom error handler

```
app.use((err, req, res, next) => {
  console.error('Error:', err.message);
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

5 MONITORING

- Monitor app in real time
- Track request rate & status codes
- Set alerts for errors
- Use with tools like PM2, Grafana, New Relic, Datadog

6 PERFORMANCE

- Track response time for each request
- Identify slow routes & backend issues
- Optimize performance

Example:

```
GET /api/data 200 250.45 ms - 1024  (Slow!)
GET /api/data 200 45.12 ms - 1024   (Good)
```

7 DEBUGGING

- Detailed logs help reproduce bugs
- Log request headers, params, body
- Easier troubleshooting

```
console.log('Headers:', req.headers);
console.log('Params:', req.params);
console.log('Body:', req.body);
```

8 ANALYTICS

- Analyze usage patterns
- Top endpoints, error rates, peak times
- Make better product & scaling decisions

Top Endpoints:

1. /api/users 45%
2. /api/products 30%
3. /api/orders 15%

CUSTOM LOG FORMAT (Example)

```
morgan.token('body', req => JSON.stringify(req.body));

const customFormat = ':method :url :status :response-time ms ' +
                     ':res[content-length] - :body';

app.use(morgan(customFormat));
```

Output Example:

```
POST /api/login 200 150 ms - {"username":"admin","password":"12345"}
```

LOG TO FILE (Optional)

```
const fs = require('fs');
const path = require('path');

app.use(morgan('combined', {
  stream: fs.createWriteStream(
    path.join(__dirname, 'access.log'),
    { flags: 'a' }
  )
}));
```

Logs will be saved in access.log file.

BEST PRACTICES

- Use 'dev' in development, 'combined' in production
- Avoid logging sensitive data (passwords, tokens)
- Rotate log files in production
- Monitor logs regularly

USEFUL MORGAN TOKENS

Token	Description	Token	Description
:method	HTTP method	:res[content-length]	Response size
:url	Request URL	:user-agent	User agent
:status	Response status	:remote-addr	IP address
:response-time	Response time in ms	:host	Request host

TOOLS & INTEGRATIONS

PM2 – Process monitoring
Winston – Advanced logging
Morgan – HTTP request logging
Grafana – Visualization
ELK Stack – Logs analytics

QUICK SUMMARY

Good logging = Better monitoring
Better monitoring = Better performance
Better performance = Happy users

Log today, fix faster, grow better!

Express

API TESTING

Test your APIs like a Pro!

POSTMAN

The most popular API testing tool

VS

THUNDER CLIENT

Fast, lightweight VS Code extension

- Test APIs Quickly
- Find & Fix Bugs
- Validate Responses
- Ensure Reliability
- Save Time
- Better Collaboration

SAMPLE API (Node.js + Express)

```
const express = require('express');
const app = express();
app.use(express.json());

let users = [
  { id: 1, name: 'John', email: 'john@example.com' },
  { id: 2, name: 'Jane', email: 'jane@example.com' }
];

// GET all users
app.get('/api/users', (req, res) => res.json(users));
// POST new user
app.post('/api/users', (req, res) => {
  const user = { id: Date.now(), ...req.body };
  users.push(user);
  res.status(201).json(user);
});
// PUT update user
app.put('/api/users/:id', (req, res) => {
  const id = parseInt(req.params.id);
  users = users.map(u => u.id === id ? { ...u, ...req.body } : u);
  res.json({ message: 'User updated', user: users.find(u => u.id === id) });
});
// DELETE user
app.delete('/api/users/:id', (req, res) => {
  const id = parseInt(req.params.id);
  users = users.filter(u => u.id !== id);
  res.json({ message: 'User deleted' });
});
app.listen(5000, () => console.log('API running on http://localhost:5000'));
```

BASIC API TEST FLOW

1. **Send Request** — Choose method, enter URL, set headers & body
2. **Get Response** — Check status code, headers, and response body
3. **Validate** — Verify data is correct (status, body, time, etc.)
4. **Save & Repeat** — Save request, automate and test again

STATUS CODES CHEAT SHEET

2xx Success
- 200 OK
- 201 Created
- 204 No Content

4xx Client Error
- 400 Bad Request
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found

5xx Server Error
- 500 Internal Server Error
- 502 Bad Gateway
- 503 Service Unavailable

GET Retrieve data from server

GET http://localhost:5000/api/users Send

Params Headers Authorization

Response (200 OK) 200 OK 15 ms 320 B

```
[
  {
    "id": 1,
    "name": "John",
    "email": "john@example.com"
  },
  {
    "id": 2,
    "name": "Jane",
    "email": "jane@example.com"
  }
]
```

POST Create new resource

POST http://localhost:5000/api/users Send

Headers Body Authorization

raw JSON

```
{
  "name": "Alice",
  "email": "alice@example.com"
}
```

Response (201 Created) 201 Created 18 ms 198 B

```
{
  "id": 1717481234567,
  "name": "Alice",
  "email": "alice@example.com"
}
```

PUT Update existing resource

PUT http://localhost:5000/api/users/1 Send

Headers Body Authorization

raw JSON

```
{
  "name": "John Updated",
  "email": "john.updated@example.com"
}
```

Response (200 OK) 200 OK 16 ms 210 B

```
{
  "message": "User updated",
  "user": {
    "id": 1,
    "name": "John Updated",
    "email": "john.updated@example.com"
  }
}
```

DELETE Remove resource

DELETE http://localhost:5000/api/users/1 Send

Headers Authorization

Response (200 OK) 200 OK 12 ms 45 B

```
{
  "message": "User deleted"
}
```

POSTMAN vs THUNDER CLIENT

POSTMAN	THUNDER CLIENT
Feature-rich	Lightweight
Collections & Environments	Built-in VS Code
Pre-request Scripts	Fast & Simple
Tests (Assertions)	Collections
Mock Servers	Environment Support
Team Collaboration	Easy for Developers
Web, Desktop & Mobile	Open Source

VS

JSON EXAMPLE

Request Body

```
{
  "name": "Bob",
  "email": "bob@example.com",
  "age": 25,
  "role": "developer"
}
```

JSON is lightweight and easy to read.

HEADERS EXAMPLE

```
Content-Type: application/json
Authorization: Bereer <token>
Accept: application/json
User-Agent: PostmanRuntime/7.36.0
```

Headers send extra information with request.

QUERY PARAMS EXAMPLE

GET /api/users?page=1&limit=10&sort=name

Key	Value
page	1
limit	10
sort	name

Query params filter or paginate data.

AUTHORIZATION EXAMPLE

Type: Bearer Token

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6Ik...

Use Bearer Token for secured APIs.

TEST SCRIPT (Postman)

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response has users", function () {
    const json = pm.response.json();
    pm.expect(json).to.be.an('array');
});
```

Automated tests ensure API works as expected.

ENVIRONMENT EXAMPLE

Variable	Initial Value	Current Value
base_url	http://localhost:5000	http://localhost:5000
token	your_token_here	your_token_here

Environments help switch between dev, test, prod.

BEST PRACTICES

- Use meaningful request names
- Validate status codes & responses
- Use environments for URLs & tokens
- Write tests for critical APIs
- Monitor API performance
- Log and handle errors properly

TIP: Test early, test often! Good API testing = Reliable Applications. | **Tools:** Postman | Thunder Client | **Formats:** JSON | **Methods:** GET | POST | PUT | DELETE | Happy Testing!

Express.js

MVC ARCHITECTURE

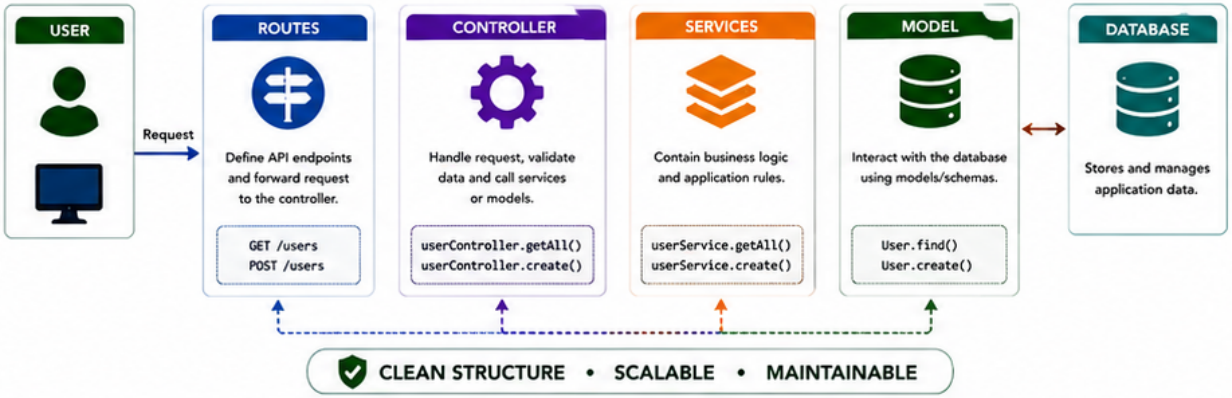
in Express.js – Clean, Scalable & Maintainable Code

WHAT IS MVC?

MVC (Model-View-Controller) is a software design pattern that separates an application into 3 interconnected components to achieve clean structure, separation of concerns and easier maintenance.

BENEFITS

- Separation of concerns
- Clean & organized code
- Easy to test and debug
- Scalable & maintainable
- Team collaboration friendly
- Reusability of components



EXAMPLE FLOW (GET /users)

```
// 1. Route
router.get('/users', userController.getAll);

// 2. Controller
exports.getAll = async (req, res) => {
  const users = await userService.getAllUsers();
  res.status(200).json({ success: true, data: users });
};

// 3. Service
exports.getAllUsers = async () => {
  return await User.find().sort({ createdAt: -1 });
};
// 4. Model (Mongoose)
conct userSchema = new Schema({
  name: String,
  email: String,
  role: String,
  createdAt: { type: Date, default: Date.now }
});
module.exports = mongoose.model('User', userSchema);
```

CLEAN STRUCTURE • SCALABLE • MAINTAINABLE

1 ROUTES

Define application endpoints and map HTTP methods to controller functions.

Example (routes/userRoutes.js)

```
conct express = require('express');
const router = express.Router();
const userController = require('../
controllers/userController');

router.get('/users', userController.getAll);
router.post('/users', userController.create);
router.get('/users/:id', userController.getById);
rootor.put('/users/:id', userController.update);
router.delete('/users/:id', userController.remove);

module.exports = router;
```

Responsibility

- Keep routes lightweight
- Only define endpoints

2 CONTROLLER

Handle incoming requests, validate data and return responses.

Example (controllers/userController.js)

```
conct userService = require('../services/
userService');

exports.getAll = async (req, res) => {
  try {
    conct users = await userService.getAllUsers();
    res.status(200).json({ success: true, data: users });
  } catch (err) {
    res.status(500).json({ success: false,
      message: err.message });
};
```

Responsibility

- Validate request
- Call services
- Send proper response

3 SERVICES

Contain business logic and reusable application functions.

Example (services/userService.js)

```
const User = require('../models/User');

exports.getAllUsers = async () => {
  return await User.find().sort({ createdAt: -1 });
};

exports.createUser = async (data) => {
  return await User.create(data);
};
```

Responsibility

- Implement business rules
- Reusable across controllers
- Keep controllers clean

4 MODEL

Define data structure and interact with the database.

Example (models/User.js)

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  role: { type: String, default: 'user' },
  createdAt: { type: Date, default: Date.now }
});
module.exports = mongouse.model('User', userSchema);
```

Responsibility

- Define schema
- Validate data
- DB operations

5 DATABASE

Persistent storage for application data.

Common Databases

MongoDB · PostgreSQL · MySQL · Redis

Responsibility

- Store data
- Ensure data integrity
- High performance

6 BUSINESS LOGIC

Rules and logic that make your application smart.

Examples

- Validate user role
- Check permissions
- Calculate totals
- Apply discounts
- Send emails
- Generate reports

Responsibility

- Solve real-world problems
- Keep logic in services layer

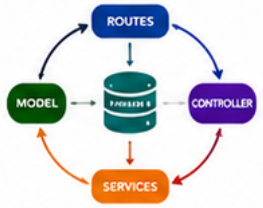
PROJECT STRUCTURE EXAMPLE

```
my-express-app/
└ src/
  └ routes/
    └ userRoutes.js
  └ controllers/
    └ userController.js
  └ services/
    └ userService.js
  └ models/
    └ User.js
  └ config/
    └ db.js
  └ app.js
  └ server.js
```

- routes/ → All route files
- controllers/ → Request handlers
- services/ → Business logic
- models/ → Database models
- config/ → Config & DB connection
- app.js → Express app setup
- server.js → Start the server

DATA FLOW IN MVC

1. User sends request (e.g., GET /users)
2. Route matches the endpoint
3. Controller handles the request
4. Service applies business logic
5. Model interacts with Database
6. Response is sent back to the user



BEST PRACTICES

- Keep controllers thin and clean
- Use services for business logic
- Validate data at controller level
- Use meaningful names
- Handle errors properly
- Keep code modular and reusable
- Write tests for services and controllers

TOOLS & TECHNOLOGIES

Express.js · MongoDB (Mongoose) · Postman · VS Code · Nodemon · ESLint (Code Quality)

WHY MVC IS BEST?

- Clean Structure
- Easy Maintenance
- Better Scalability
- Faster Development
- Team Collaboration
- Production Ready

Write Clean Code. Build Scalable Apps.

Express.js

CONTROLLERS

in Express.js – Handle Request, Business Logic & Response

Controllers act as the brain of your application.
They receive requests, process data using business logic and send responses.

REQUEST
Client sends HTTP request

→

CONTROLLER
Handle request, apply business logic (call services/models)

→

RESPONSE
Send appropriate response back to client

Example Controller (userController.js)

```
1  const userService = require('../services/userService');
2
3  // GET all users
4  exports.getAllUsers = async (req, res) => {
5    try {
6      const users = await userService.getAll();
7      res.status(200).json({ success: true, data: users });
8    } catch (error) {
9      res.status(500).json({ success: false, message: error.message });
19   }
10 };
11
13 // CREATE a new user
14 exports.createUser = async (req, res) => {
16   const user = await userService.create(req.body);
16   res.status(201).json({ success: true, data: user });
17 };
```

WHY CONTROLLERS?

- Keep routes clean & small
- Separate business logic
- Reusable & maintainable code
- Easy to test and debug
- Follow Single Responsibility Principle (SRP)
- Better scalability

1 CONTROLLER

A controller is a function or a group of functions that handle HTTP requests.

Example
```
exports.getUser = (req, res) => { ... }
```

- Receives request from routes
- Calls service/model for data
- Sends response to client

2 REQUEST

Controller receives request object (req) with params, body, query, headers and more.

Example
```
// req example
req.params.id, req.body, req.query
```

- Contains client data
- Used for validation
- Drives business logic

3 BUSINESS LOGIC

Apply business rules, validate data, call services or models, perform operations.

Example
```
const user = await userService.find(id);
if (!user) throw new Error('Not found');
```

- Keep logic in services/models
- Controllers stay thin
- Easy to change & test

4 RESPONSE

Send proper HTTP response with status code and JSON data.

```
res.status(200).json({
  sucess: true,
  data: user
});
```

- Send success or error
- Use correct status codes
- Consistent response format

5 FUNCTIONS

Controllers are made of functions. Each function handles a specific task.

```
exports.getUser = () => {}
exports.createUser = () => {}
```

- One function = One task
- Improves readability
- Easy to maintain

6 CRUD OPERATIONS

Controllers handle CRUD operations (Create, Read, Update, Delete).

```
CREATE  -> POST
READ    -> GET
UPDATE  -> PUT/PATCH
DELETE  -> DELETE
```

- Standard in every API
- Map to HTTP methods
- Easy for clients to use

7 API LOGIC

Encapsulate API specific logic like validation, parsing, transforming, etc.

```
Validate -> Process -> Respond
```

- Sanitize & validate input
- Transform data if needed
- Handle errors gracefully

8 REUSABLE

Controllers and their logic can be reused across routes and modules.

Use same function in multiple routes

- DRY (Don't Repeat Yourself)
- Modular & reusable code
- Better developer experience

CONTROLLER CRUD EXAMPLE (User)

Method	Endpoint	Controller Function	Description
GET	/api/users	getAllUsers	Get all users
GET	/api/users/:id	getUserById	Get user by ID
POST	/api/users	createUser	Create new user
PUT	/api/users/:id	updateUser	Update user by ID
DELETE	/api/users/:id	deleteUser	Delete user by ID

ROUTE → CONTROLLER FLOW

1. Client sends request
2. Route matches the endpoint
3. Controller function is called
4. Business logic is executed
5. Response is sent back

BEST PRACTICES

- Keep controllers thin (call services)
- Validate incoming data
- Use async/await for async code
- Handle errors using try/catch
- Return consistent response format
- Use meaningful status codes
- Avoid duplicating logic

CONTROLLER STRUCTURE EXAMPLE

```
controllers/
  userController.js
```

```
exports.getAllUsers = async () => {};
exports.getUserById = async () => {};
exports.createUser = async () => {};
exports.updateUser = async () => {};
exports.deleteUser = async () => {};

module.exports = exports;
```

EXAMPLE ERROR HANDLING

```
exports.getUserById = async (req, res) => {
  try {
    const user = await userService.findById(req.params.id);
    if (!user) {
      return res.status(404).json({ success: false, message: "User not found" });
    }
    res.status(200).json({ success: true, data: user });
  } catch (error) {
    console.error(error);
    res.status(500).json({ success: false, message: "Server Error" });
};
```

Controllers make your API organized, maintainable and scalable.

Clean Code | Separation of Concerns | Reusable | Testable | Scalable

Good Controllers = Strong APIs!

Express.js

MONGOOSE

MODELS

in Express.js with Mongoose

Models define the structure of your data and interact with MongoDB.

MODEL	SCHEMA	COLLECTION	DOCUMENT	VALIDATION	METHODS	DATABASE	MONGOOSE
Interface to the database. Created from a Schema.	Blueprint of your data structure.	Group of documents in MongoDB.	Single record in a collection. (JSON-like)	Ensure data integrity with built-in rules.	Custom or built-in functions.	MongoDB stores your collections.	ODM for MongoDB in Node.js. Simple & powerful.

EXAMPLE MODEL (User.js)

```
const mongoose = require('mongoose');

// 1. Define Schema
const userSchema = new mongoose.Schema({
  name: { type: String, required: true, trim: true },
  email: { type: String, required: true, unique: true, lowercase: true },
  password: { type: String, required: true, minlength: 6 },
  age: { type: Number, min: 0 },
  createdAt: { type: Date, default: Date.now }
}, { timestamps: true });

// 2. Add Custom Method
userSchema.methods.getFullName = function() {
    return this.name;
};

// 3. Create Model
const User = mongoose.model('User', userSchema);
module.exports = User;
```

WHY USE MODELS?

- Organized & clean structure
- Reusability of code
- Data validation & integrity
- Built-in methods & middleware
- Easy CRUD operations
- Better maintainability
- Powerful querying
- Works seamlessly with MongoDB

1 MODEL

A Model is a constructor compiled from a Schema. It represents a collection in the database.

Example

```
const User = mongoose.model("User", userSchema);
```

- Provides an interface to the database.
- Used to create, read, update, delete data.
- Each model maps to a collection.

2 SCHEMA

Schema defines the structure of documents in a collection. It includes fields, types, validation & defaults.

Example

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, unique: true },
  age: { type: Number, min: 0 }
});
```

- Blueprint for your documents.
- Defines field types & rules.
- Enforces data consistency.

3 COLLECTION

A collection is a group of documents. Mongoose creates it automatically (plural form of model name).

Example

```
// User model -> users collection
User.create({ name: "John" });
```

- Stored in MongoDB.
- NoSQL – schema less at DB level.
- Scales horizontally.

4 DOCUMENT

A document is a single record (JSON-like object) in a collection.

Example

```
{
  "_id": ObjectId("662.."),
  "name": "John",
  "email": "john@example.com",
  "age": 25,
  "createdAt": "2024-05-26T10:00:00Z"
}
```

- Represents real data.
- Each document has a unique _id.
- Flexible structure.

5 VALIDATION

Validate data before saving to the database using built-in validators.

Example

```
email: { type: String,
   required: true,
   unique: true,
   match: /.+@.+\..+/ }
age: { type: Number, min: 0, max: 120 }
```

- required, unique, min, max
- match (regex), enum, default
- Custom validators supported.

6 METHODS

Add custom instance or static methods for reusability.

Example (Instance Method)

```
userSchema.methods.getFullName =
  function() {
    return this.name;
};
```

- Instance methods: on documents
- Static methods: on model
- Promotes DRY principle.

7 DATABASE (MongoDB)

MongoDB stores your data as collections & documents.

Example

MongoDB → Database

- NoSQL Database
- High performance
- Flexible & scalable

8 MONGOOSE

Mongoose is an ODM (Object Data Modeling library) for MongoDB and Node.js.

Example

```
const mongoose = require('mongosse');
mongoose.connect(uri, options);
```

- Simplifies MongoDB operations.
- Built-in query helpers.
- Middleware & hooks support.

COMMON SCHEMA TYPES

Type	Description
String	Text data
Number	Numeric data
Boolean	true / false
Date	Date & Time
Array	List of values
ObjectId	Reference to other document
Mixed	Any type of data

COMMON VALIDATORS

Validator	Usage
required: true	Field is mandatory
unique: true	Must be unique
min / max	For Number & String length
match: /regex/	Pattern validation
enum: [..]	Allowed values
default: value	Default value
trim: true	Removes spaces

CRUD OPERATIONS WITH MODEL

Operation	Mongoose Method	Example
Create	Model.create(data)	User.create({ name: 'John' })
Read One	Model.findById(id)	User.findById(id)
Read Many	Model.find(filter)	User.find({ age: { $gte: 18 } })
Update One	Model.findByIdAndUpdate(id, data)	User.findByIdAndUpdate(id, { name: 'Mike' })
Delete One	Model.findByIdAndDelete(id)	User.findByIdAndDelete(id)
Delete Many	Model.deleteMany(filter)	User.deleteMany({ age: { $lt: 18 } })

MODEL MIDDLEWARE (HOOKS)

Hook	When it Executes
pre('save')	Before saving a document
post('save')	After saving a document
pre('find')	Before find query
post('find')	After find query
pre('remove')	Before removing
post('remove')	After removing

PROJECT STRUCTURE EXAMPLE

```
my-express-app/
├── models/
│   └── User.js              # Model (Schema + Methods)
├── routes/
│   └── userRoutes.js        # Routes
├── controllers/
│   └── userController.js    # Controller (Business Logic)
├── services/
│   └── userService.js       # Services / Business Logic
├── config/
│   └── db.js                # DB Connection
├── app.js                   # App Entry Point
└── package.json
```

TIP: Always keep models in a separate folder. | Validate data at model level. | Use methods to keep controllers clean. | Follow DRY & SRP principles. | Good Models = Strong & Scalable Applications!

Express.js

SERVICES

in Express.js – Keep Business Logic Separate

Services contain business logic and can be used by Controllers, Routes or other Services. They make your code clean, reusable and maintainable.

BUSINESS LOGIC	API CALLS	DATABASE LOGIC	REUSABLE CODE	HELPER FUNCTIONS	MODULES	CLEAN CODE
Handle core application logic	Make external API requests safely	Interact with Database via Models	Reuse in multiple controllers or services	Small utility functions & helpers	Organized in modules for scalability	Separation of concerns = Clean Code

Example Service: userService.js

```
const axios = require('axios');
const User = require('../models/User');

// Get all users
const getAllUsers = async () => {
  return await User.find().sort({ createdAt: -1 });
};
// Get user by ID
const getUserById = async (id) => {
  const user = await User.findById(id);
  if (!user) throw new Error('User not found');
  return user;
};
// Create new user
const createUser = async (data) => {
  const user = new User(data);
  return await user.save();
};
// Call external API
const getExternalData = async () => {
  const res = await axios.get('https://jsonplaceholder.typicode.com/posts?_limit=5');
  return res.data;
};
module.exports = { getAllUsers, getUserById, createUser, getExternalData };
```

WHY USE SERVICES?

- Keeps controllers thin
- Easy to test
- Reusable across project
- Better maintainability
- Separation of concerns
- Scalable architecture
- Clean and professional code

1 SERVICES

Services contain the core business logic of the application.

Example

```
// userService.js
const getAllUsers = async () => {
  return await User.find();
}
```

- Not tied to any route
- Can be used anywhere
- Focus on doing the job

2 BUSINESS LOGIC

All complex logic and calculations should live in services.

Example

```
const totalPrice = items.reduce(
  (sum, item) => sum + item.price,
  0
);
```

- Validate data
- Apply rules
- Process information

3 API CALLS

Services can call external APIs and return data.

Example

```
const res = await axios.get(
  'https://api.example.com/data'
);
return res.data;
```

- Use axios/fetch
- Handle errors
- Return useful data

4 DATABASE LOGIC

Perform all DB operations inside services.

Example

```
const user = await User.findById(id);
if (!user) throw new Error('Not found');
return user;
```

- CRUD operations
- Aggregation
- Transactions

5 REUSABLE CODE

Write once, use multiple times in project.

Example

```
// Used in many controllers
const users = await userService
  .getAllUsers();
```

- Reduce duplication
- Easy updates
- Improves productivity

6 HELPER FUNCTIONS

Small utility functions to help with tasks.

Example

```
const formatDate = (date) =>
  new Date(date).toISOString(
  .split('T')[0];
```

- Format data
- Generate tokens
- Send emails

7 MODULES

Organize services in modules for better structure.

Example

```
/services
  ├── userService.js
  ├── orderService.js
  └── paymentService.js
```

- Separation by feature
- Easy to manage
- Scalable project

8 CLEAN CODE

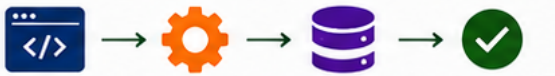
Services help follow clean architecture principles.

Example

```
Controller → Service → Model
(Flow stays clean)
```

- Thin controllers
- Readable code
- Easy to test & debug

SERVICE LAYER FLOW



Controller is like a waiter, Service is like a chef.
Controller takes order, Service prepares it, Model/DB gets the data.

EXAMPLE USAGE IN CONTROLLER

```
const userService = require('../services/userService');

exports.getUsers = async (req, res) => {
  try {
    const users = await userService.getAllUsers();
    res.status(200).json({ success: true, data: users });
  } catch (error) {
    res.status(500).json({ success: false, message: error.message });
  }
};
```

PROJECT STRUCTURE EXAMPLE

```
my-express-app/
├── controllers/
│   └── userController.js
├── services/
│   └── userService.js  ← Business Logic
├── models/
│   └── User.js
├── routes/
│   └── userRoutes.js
├── utils/
│   └── helpers.js
└── app.js
```

Handles request & response

Business logic & operations

Database schema & models

Helper functions & utilities

QUICK SUMMARY

Topic	Key Point
Services	Contain business logic
Business Logic	Process rules & data
API Calls	Call external services
Database Logic	Interact with DB via Models
Reusable Code	Use across multiple places
Helper Functions	Small reusable utilities
Modules	Organize by feature
Clean Code	Separation of concerns

PRO TIP: Keep your controllers small and let services do the heavy lifting!

Follow Best Practices: ✅ Single Responsibility ✅ DRY Principle ✅ Clean Code ✅ Meaningful Names

Great Services = Strong & Scalable Applications!

Express

UTILITY FUNCTIONS

Reusable Helpers & Utils for Clean and Efficient Code

Utility functions are small, reusable pieces of code that make our application clean, modular and easy to maintain.

UTILS	HELPERS	FORMATTER	VALIDATOR	LOGGER	CONSTANTS	REUSABLE FUNCTIONS	CONFIGURATION
Small functions with big impact	Help in common tasks and operations	Format data in consistent way	Validate data before using it	Log events, errors and information	Store constant values in one place	Write once, use anywhere	Manage app configuration easily

Example: utils/index.js

```
const crypto = require('crypto');
const moment = require('moment');

// Formatter
const formatDate = (date) =>
  moment(date).format('YYYY-MM-DD HH:mm:ss');

// Validator
const isEmail = (email) =>
  /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);

// Logger
const logger = (level, message, meta = {}) => {
  const time = formatDate(new Date());
  console(level)(`[${time}] ${level.toUpperCase(): ${message}`, meta);
};
// Constant
const APP_CONSTANTS = {
  DEFAULT_PAGE: 1,
  PAGE_LIMIT: 10,
  TOKEN_EXPIRE: '7d'
};
module.exports = {
  formatDate,
  logger,
}; APP_CONSTANTS
```

WHY USE UTILITY FUNCTIONS?

- Keeps code DRY (Don't Repeat Yourself)
- Improves readability
- Easier maintenance
- Reusable across project
- Reduces code duplication
- Centralized common logic
- Faster development
- Clean and professional code

1 UTILS (GENERAL)

General utility functions used in many places.

Example

```
const toTitleCase = (str) =>
  str.replace(/\w\S*/g, (txt) =>
    txt.charAt(0).toUpperCase() +
    txt.substr(1).toLowerCase())
);
```

- String, Number, Array helpers
- Common operations
- Reusable everywhere

2 HELPERS

Helper functions to simplify complex operations.

Example

```
const getFullName = (user) =>
'${user.firstName} ${user.lastName}';
```

- Small & focused functions
- Make code clean
- Easy to test

3 FORMATTER

Format data in consistent and user-friendly way.

Example

```
const formatCurrency = (amount) =>
  new Intl.NumberFormat('en-IN', {
    style: 'currency',
    currency  'INR'
  }).format(amount);
```

- Date, Time, Currency, etc.
- Consistent output
- Better user experience

4 VALIDATOR

Validate data before processing or saving it.

Example

```
const isPhone = (phone) =>
  /^[6-9]\d{9}$/.test(phone);
```

- Validate email, phone, input
- Prevent invalid data
- Improve data integrity

5 LOGGER

Log important info, warnings and errors.

Example

```
logger('error', 'Something
went wrong!', { userId: 123 });
```

- Info, Warn, Error logs
- Debugging easy
- Track application flow

6 CONSTANTS

Store constant values in a single place.

Example

```
module.exports = {
  STATUS: {
    ACTIVE: 1,
    INACTIVE: 0
} }
```

- Avoid magic numbers
- Easy to update
- Used across app

7 REUSABLE FUNCTIONS

Write once, use multiple times.

Example

```
const generateId = () =>
crypto.randomBytes(8).toString('hex');
```

- Avoid repetition
- Modular & reusable
- Save development time

8 CONFIGURATION

Manage configuration values in one place.

Example

```
module.exports = {
  port: process.env.PORT || 5000,
  dbUrl: process.env.MONGO_URI,
  nodeEnv: process.env.NODE_ENV
};
```

- Environment based config
- Secure and flexible
- Easy to manage

MORE UTILITY EXAMPLES

generateId	`() => crypto.randomUUID()`	Generate unique ID
truncateText	`(text, len) => text.slice(0, len) + '...'`	Truncate long text
sanitize	`(str) => str.replace(/<[^>]*>/g, '')`	Remove HTML tags
sleep	`(ms) => new Promise(res => setTimeout(res.`	Delay execution
pick	`(obj, keys) => keys.reduce((o, k) => (o[k] = obj[k], o), {})`	Pick object fields

FOLDER STRUCTURE EXAMPLE

```
my-express-app/
├─ utils/
│  ├─ index.js
│  ├─ formatter.js
│  ├─ validator.js
│  ├─ logger.js
│  ├─ constants.js
│  └─ config.js
├─ controllers/
├─ services/
├─ routes/
└─ app.js
```

All utility files inside /utils

Keep your utils organized

Easy to import & use anywhere

HOW TO USE UTILS

```
// Import utils
const { formatDate, isEmail, logger, APP_CONSTANTS }
  = require('./utils');

app.post('/user', (req, res) => {
  const { email } = req.body;

  if (!isEmail(email)) {
    logger('warn', 'Invalid email provided', { email });
    return res.status(400).json({ message: 'Invalid email' });
  }

  const now = formatDate(new Date());
  logger('info', 'New user request received', { time: now });
  res.status(200).json({ message: 'Valid request' });
});
```

CONFIG EXAMPLE (.env)

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/myapp
NODE_ENV=development
LOG_LEVEL=info
JWT_SECRET=your_secret_key

// config.js
module.exports = {
  port: process.env.PORT,
  dbUrl: process.env.MONGO_URI,
  env: process.env.NODE_ENV,
  logLevel: process.env.LOG_LEVEL,
  jwtSecret: process.env.JWT_SECRET
};
```

BEST PRACTICES

- Keep utility functions small and focused
- Use meaningful names
- Follow single responsibility principle
- Write reusable and testable code
- Keep all constants in one file
- Use environment variables for config
- Add comments for complex logic
- Keep utils independent of other modules

TIP: Utilities may be small, but they make your big project strong! | DRY Principle | Reusable Code | Clean Code | Easy Maintenance | Better Scalability | **Good Utilities = Great Applications!**

Express.js

PROJECT STRUCTURE

A Clean & Scalable Structure for Express.js Applications

A well-organized project structure makes your code easy to understand, maintain and scale.

PROJECT STRUCTURE

```
my-express-app/
├ src/
│  ├── routes/
│  ├── controllers/
│  ├── models/
│  ├── middleware/
│  ├── services/
│  └── utils/
├ .env
├ .gitignore
├ package.json
└ app.js
```

FOLDER / FILE	PURPOSE
src/	Root directory for all source code. Keeps everything organized.
routes/	Define all API routes and endpoints. Handles HTTP requests mapping.
controllers/	Handle request, response and application logic. Keep controllers thin and clean.
models/	Define database models (Schemas). Interact with the database.
middleware/	Custom middlewares (auth, error handling, etc). Executes before route handlers.
services/	Business logic layer. Reusable code and external API calls.
utils/	Helper functions, formatters, validators, constants, and other utilities.
app.js	Entry point of the application. Initializes Express and connects all parts.

app.js (Entry Point)

```
1  const express = require('express');
2  const app = express();
3  const cors = require('cors');
4  const morgan = require('morgan');
5  dotenv = require('dotenv').config();
6
7  // Middlewares
8  app.use(cors());
9  app.use(express.json());
10 app.use(morgan('dev'));
11
12 // Routes
13 const userRoutes = require('./src/routes/userRoutes');
14 const authRoutes = require('./src/routes/authRoutes');
15
16 app.use('/api/users', userRoutes);
17 app.use('/api/auth', authRoutes);
18
19 // Global Error Handler
20 const errorHandler = require('./src/middleware/errorHandler');
21 app.use(errorHandler);
22
23 const PORT = process.env.PORT || 5000;
24 app.listen(PORT, () => console.log('Server running on port ${PORT}'));
```

WHY THIS STRUCTURE?

- Separation of concerns
- Easy to maintain
- Scalable for large projects
- Team friendly
- Reusable code
- Easy testing & debugging
- Clean and professional

FLOW OF A REQUEST

Client Request
↓
Routes
↓
Middleware
↓
Controller
↓
Services
↓
Models (DB)
↓
Response

COMPLETE PROJECT TREE

```
my-express-app/
├ src/
│  ├ routes/
│  │  ├ userRoutes.js
│  │  ├ authRoutes.js
│  │  └ index.js
│  ├ controllers/
│  │  └ userController.js
│  ├ models/
│  │  └ User.js
│  ├ middleware/
│  │  └ auth.js
│  ├ services/
│  │  └ userService.js
│  ├ utils/
│  │  └ logger.js
│  └ app.js
├ .env
├ package.json
└ .gitignore
```

Everything inside src/ is application source code.

Each folder has a defined responsibility.

app.js is the heart of the application.

This structure is ready for production!

WHAT EACH FOLDER MAY CONTAIN

routes/	controllers/	models/	middleware/	services/	utils/
userRoutes.js	userController.js	User.js	auth.js	userService.js	logger.js
authRoutes.js	authController.js	Product.js	errorHandler.js	emailService.js	validator.js
productRoutes.js	productController.js	Order.js	Ordee.js	paymentService.js	formatter.js
index.js		index.js	notFound.js		constants.js

GOOD STRUCTURE = CLEAN CODE = SUCCESSFUL PROJECT | Keep Controllers Thin | Move Logic to Services | Reusability is the Key | Follow Best Practices | Build Scalable Apps

socket.io

SOCKET.IO

Real-Time Communication Made Easy

Socket.IO enables real-time, bi-directional and event-based communication between clients and server.

HOW SOCKET.IO WORKS

SERVER (Node.js + Socket.IO) — server.js

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = new Server(server, { cors: { origin: "*" } });

// When a client connects
io.on('connection', (socket) => {
        .log('✅ New client connected', socket.id);

    // Listen for chat message
    socket.on('sendMessage', (data) => {
      console.log('Message:', data);
      // Broadcast to all clients (including sender)
      io.emit('receiveMessage', data);
    });

    // Handle disconnect
    socket.on('disconnect', () => {
        console.log('❌ Client disconnected', socket.id);
    });
});

server.listen(3000, () => {
  console.log('🚀 Server running on port 3000');
});
```

CLIENT (Browser) — client.js

```
const socket = io('http://localhost:3000');
const form = document.getElementById('form');
const input = document.getElementById('input');
const messages = document.getElementById('messages');

// Connection
socket.on('connect', () => {
   console.log('✅ Connected to server:', socket.id);
});

// Listen for new messages
socket.on('receiveMessage', (data) => {
    const item = document.createElement('li');
    item.textContent = data.message;
    messages.appendChild(item);
    window.scrollTo(0, document.body.scrollHeight);
});
// Send message
form.addEventListener('submit', (e) => {
  e.preventDefault();
  const data = { message: input.value, time: new Date().toLocaleTimeString() };
  socket.emit('sendMessage', data);
  input.value = '';
});
```

Real-Time — Instant communication

Bi-Directional — Two-way data flow

Event Based — Communicate using events

Broadcast — Send message to many

Extensible — Middleware & adapters

EMIT

Send an event from client to server.

```
socket.emit('event', data);
```

Example

```
socket.emit('sendMessage',
{ message: 'Hello' });
```

ON

Listen for an event.

```
socket.on('event', (data) => });
```

Example

```
socket.on('receiveMessage',
(data) => { ... });
```

BROADCAST

Send event to all connected clients.

```
io.emit('event', data);
```

Example

```
io.emit('receiveMessage',
data);
```

TO / IN

Send event to specific room or client.

```
io.to(roomId).emit('event', data);
socket.to(userId).emit('event', data);
```

Example

```
io.to('room1').emit('message', data);
```

ROOMS

Group clients in a room and communicate.

```
socket.join('room1');
io.to('room1').emit(...);
```

Example

```
socket.join('room1');
io.to('room1').emit('msg', data);
```

NAMESPACES

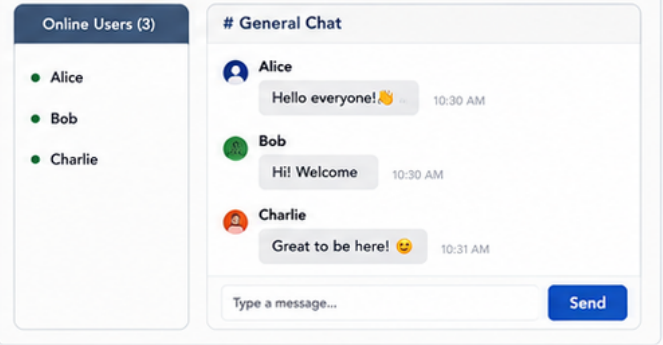
Create isolated communication channels.

```
const chat = io.of('/chat');
```

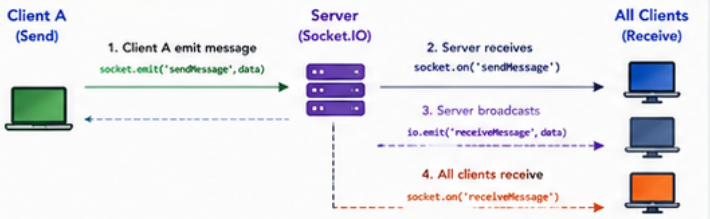
Example

```
const chat = io.of('/chat');
chat.emit('message', data);
```

REAL-TIME CHAT APPLICATION (DEMO)



EVENT FLOW (CHAT EXAMPLE)



PROJECT STRUCTURE

```
my-chat-app/
├── server.js
├── public/
│   ├── index.html
│   └── client.js
└── package.json
```

- server.js → Main server file
- public/ → Frontend files
- index.html → Chat UI
- client.js → Client side logic
- package.json → Project dependencies

DEPENDENCIES (package.json)

```
{
  "name": "socketio-chat-app",
  "version": "1.0.0",
  "description": "Real-time chat app using Socket.IO",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "socket.io": "^4.7.2"
  },
  "devDependencies": {
    "nodemon": "^3.0.1"
  }
}
```

BEST PRACTICES

- ✓ Use rooms for group chats.
- ✓ Use acknowledgements for important events.
- ✓ Handle disconnects properly.
- ✓ Validate and sanitize data.
- ✓ Use middlewares for authentication.
- ✓ Keep events and data structure consistent.
- ✓ Monitor and handle errors.

Socket.IO = Real-Time Power for Web Apps | Real-Time Updates | Bi-Directional Events | Scalable & Fast | Rooms & Namespaces | Easy to Integrate | Build Real-Time. Build Better.

WebSockets

WEBSOCKETS

Real-Time Communication Made Simple

WebSockets provide a persistent, full-duplex connection between client and server for real-time data exchange.

HOW WEBSOCKET CONNECTION WORKS

CLIENT (Browser)

1. HTTP Handshake (Upgrade Request) →
2. ← HTTP 101 Switching Protocols
3. WebSocket Connection Established
4. ↔ Bidirectional Communication (Real-Time)

SERVER (Node.js)

SERVER (Node.js)

```
const express = require('express');
const http = require('http');
const WebSocket = require('ws');

const app = express();
const server = http.createServer(app);
const wss = new WebSocket.Server({ server });

wss.on('connection', (ws) => {
  console.log('Client connected');
  ws.on('message', (message) => {
    console.log('Received:', message);
    ws.send('Server: ${message}');
  });
  ws.send('Welcome to WebSocket Server!');
});

server.listen(8080, () => {
  console.log('WebSocket server running on port 8080');
});
```

KEY FEATURES

- Persistent Connection
- Full-Duplex Communication
- Handshake (HTTP Upgrade)
- Real-Time Data Exchange
- Low Latency
- Efficient & Lightweight
- Supports Web & Mobile

CONNECTION

Establish a persistent connection between client and server.

`new WebSocket('ws://server:8080')`

HANDSHAKE

Uses HTTP Upgrade mechanism to switch to WebSocket protocol.

`HTTP 101 Switching Protocols`

BIDIRECTIONAL

Both client and server can send messages at any time.

`ws.send()` and `ws.on('message')`

REAL-TIME

Ideal for applications that need instant updates and low latency.

`Live data in real-time`

EVENTS

Listen and respond to events on the WebSocket connection.

`ws.on('message')`

LIVE DATA

Stream live updates like stock prices, notifications, and more.

`Continuous data stream`

PUSH NOTIFICATION

Send data from server to client instantly without polling.

`Instant push updates`

CLIENT (Browser)

```
const ws = new WebSocket('ws://localhost:8080');
ws.onopen = () => {
  console.log('Connected to Server');
  ws.send('Hello Server!');
};
ws.onmessage = (event) => {
  console.log('Message from Server:', event.data);
};
ws.onclose = () => {
  console.log('Disconnected from Server');
};
// Send a message
function sendMessage(message) {
  ws.send(message);
}
```

REAL-WORLD USE CASES

- Live Chat Applications
- Stock Market Updates
- Live Sports Scores
- Online Gaming
- Collaborative Tools
- IoT Device Communication

WEBSOCKET FLOW

CLIENT ↔ SERVER

- CONNECTION →
- MESSAGES (REAL-TIME) ↔
- DISCONNECTION ⇢

Persistent • Full-Duplex • Real-Time

BEST PRACTICES

- Use secure WebSockets (wss://)
- Handle connection errors gracefully
- Implement reconnection logic
- Validate and sanitize incoming data
- Keep messages lightweight
- Close connections when not needed

Real-Time Communication | Low Latency | Full-Duplex | Persistent Connection | Better User Experience | Build Real-Time Applications

PERFORMANCE OPTIMIZATION

Write Better Code. Use Resources Smartly. Deliver Fast Response.

Performance optimization improves speed, scalability and user experience by using best practices and efficient resource management.

FAST RESPONSE

Better Performance, Better Experience

Example: Optimization in Action

```
const express = require('express');
const compression = require('compression');
const NodeCache = require('node-cache');
const app = express();

app.use(compression()); // Enable Gzip Compression
const cache = new NodeCache({ stdTTL: 60 }); // Cache for 60s

app.get('/data', async (req, res) => {
  const cached = cache.get('data');
  if (cached) return res.json({ source: 'cache', data: cached });
  const data = await getDataFromDB(); // Expensive Operation
  cache.set('data', data);
  res.json({ source: 'db', data });
});
```

OPTIMIZATION GOAL

- Reduce Response Time
- Use Resources Efficiently
- Handle More Users
- Reduce Server Load
- Improve Scalability
- Better User Experience

1 COMPRESSION

Compress response data before sending to client.

```
app.use(compression());
```

- Reduces data size
- Speeds up response
- Save bandwidth

2 CACHING

Store frequently used data in memory to reduce database calls.

```
const cache = new NodeCache({
  stdTTL: 60 // 60 seconds
});
```

- Faster data retrieval
- Reduces database load
- Improves throughput

3 ASYNC CODE

Use asynchronous code to handle more requests without blocking.

```
app.get('/', async (req, res) => {
  const data = await getData();
  res.send(data);
});
```

- Non-blocking I/O
- Better concurrency
- Higher performance

4 CLUSTER

Use multiple CPU cores to handle more connections simultaneously.

```
const cluster = require('cluster');
if (cluster.isMaster) {
  cluster.fork();
}
```

- Utilize multi-core CPUs
- Increases performance
- Better for high traffic

5 MEMORY

Manage memory usage and prevent memory leaks in your application.

```
process.memoryUsage();
global.gc && global.gc();
```

- Lower memory usage
- Prevents memory leaks
- Keeps app stable

6 PROFILING

Profile your application to find bottlenecks and slow sections.

```
node --inspect app.js
chrome://inspect
```

- Find performance issues
- Analyze CPU & Memory
- Optimize bottlenecks

7 OPTIMIZATION

Apply best practices and optimize code, queries & resources.

```
• Optimize DB queries
• Use indexes
• Minimize middleware
```

- Efficient code
- Better scalability
- Improved performance

8 FAST RESPONSE

Deliver fest and consistent response to users.

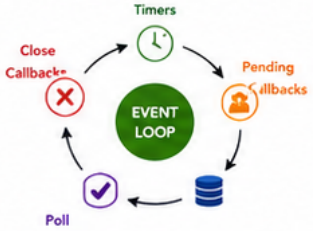
```
res.set('X-Powered-By', '');
res.send(data);
```

- Low latency
- Happy users
- Better SEO & Ranking

PERFORMANCE TIPS

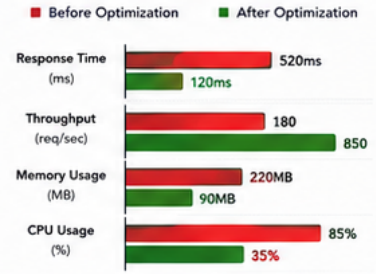
- **Use Compression (Gzip/Brotli)** — Reduce payload size
- **Enable Caching** — Cache responses, queries, and objects
- **Write Async Code** — Avoid blocking the event loop
- **Use Cluster / PM2** — Utilize all CPU cores
- **Optimize Database** — Use indexes, projection, and aggregation
- **Limit & Paginate** — Return only needed data
- **Monitor & Profile** — Use tools to find bottlenecks
- **Keep Dependencies Updated** — For performance & security

EVENT LOOP (NODE.JS)



Non-blocking operations keep the event loop free to handle more requests.

PERFORMANCE BEFORE vs AFTER



⭐ Optimization = Faster app, Happy users, Scalable system!

TOOLS FOR PROFILING & MONITORING

Chrome DevTools	Profile frontend & backend performance
clinic.js	Great for Node.js performance analysis
0x (ZeroX)	Flamegraph for CPU profiling
PM2 Monitoring	Monitor memory, CPU, and logs
New Relic	APM for production monitoring
Datadog	Infrastructure & application monitoring

BEST PRACTICES

- Always use compression
- Cache where possible
- Avoid blocking code
- Optimize database queries
- Use connection pooling
- Monitor in production
- Scale horizontally
- Test performance regularly

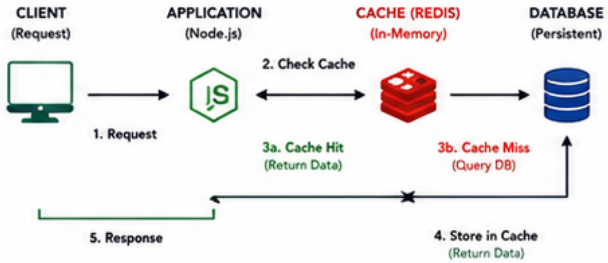
Optimize Code | Use Resources Smartly | Improve Performance | Scale Effortlessly | Happy Users | Fast Response, Better Business!

CACHING With Redis

Faster Apps, Better Experience

Caching stores frequently used data in memory so your application can respond faster and handle more users with less load.

HOW CACHING WORKS



Cache Hit ✅
Data found in cache.
Fast response.

Cache Miss ❌
Data not in cache.
Fetch from DB, store in cache.

REDIS IN ACTION (Node.js)

```
const redis = require('redis');
const client = redis.createClient();
await client.connect();

// Set data with TTL (Time To Live)
await client.setEx('user:123', 60, JSON.stringify({
    id: 123,
    name: 'CodeSardar',
    role: 'Admin'
}));

// Get data
const data = await client.get('user:123');
if (date) {
    consolo.log('✅ Cache Hit:'; JSON.parse(data));
} else {
    consolo.log('❌ Cache Miss: Fetch from DB');
}
// Delete data
await client.del('user:123');
await client.quit();
```

WHY USE REDIS?

- **Extreme Performance** — In-memory data access is very fast.
- **High Scalability** — Handles thousands of requests per second.
- **Versatile Data Structures** — Strings, Hashes, Lists, Sets, Sorted Sets, etc.
- **TTL Support** — Set expiration time for automatic cleanup.
- **Reliable & Persistent** — Supports persistence and replication.

KEY CONCEPTS

- **Redis** — Open-source in-memory data structure store, used as a cache, database, and message broker.
- **Memory Cache** — Stores data in RAM for ultra-fast read/write operations.
- **TTL (Time To Live)** — Set expiration time for keys. Data is auto-removed after TTL.
- **Cache Hit** — Requested data found in cache. Fast response, no DB query.
- **Cache Miss** — Requested data not found in cache. Fetch from DB and store in cache.
- **Session Store** — Store user sessions in Redis for fast and scalable session management.

CACHE FLOW EXAMPLE

CACHE HIT FLOW

1. Client requests data (GET /user/123)
2. Application checks Redis
3. Data found in Redis (Cache Hit)
4. Return data to client (Fast Response)

CACHE MISS FLOW

1. Client requests data (GET /user/123)
2. Application checks Redis
3. Data not found (Cache Miss)
4. Fetch from Database
5. Store in Redis with TTL
6. Return data to client (Slightly Slower)

EXAMPLE: USER PROFILE CACHE

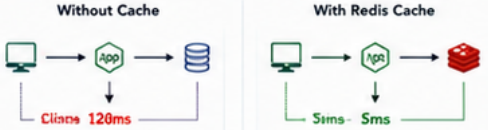
Step	Action	Detail
1	Request	User requests /user/123
2	Check Cache	Check Redis for key: user:123
3a	Cache Hit	Return data from Redis (fast)
3b	Cache Miss	Query DB for user data
4	Store in Cache	Store data in Redis with TTL (e.g., 300s)
5	Response	Send data back to user

Example Key with TTL

```
SET user:123 "{\"id\":123,\"name\":\"CodeSardar\"}" EX 300

# Key will expire automatically after 300 seconds (5 minutes)
```

PERFORMANCE IMPACT



~95% Faster Response!

BEST PRACTICES

- Cache only what is frequently read.
- Set appropriate TTL for auto-expiration.
- Use meaningful key naming conventions.
- Invalidate or update cache when data changes.
- Monitor Redis memory usage.
- Use Redis Cluster for high availability & scalability.

COMMON USE CASES

- API Response Caching
- User Session Store
- Product Data Caching
- Leaderboard & Rankings
- Rate Limiting & Counters
- Real-time Analytics
- Notification Queues

AT A GLANCE

Technology	Redis
Type	In-Memory Data Store
Language Support	Multi (Node.js, Python, Java, etc.)
Default Port	6379
Data Persistence	Optional (RDB, AOF)
Best For	Speed, Scalability, Real-time Apps

KEY BENEFITS

- Blazing Fast
- Low Latency
- Highly Scalable
- Cost Effective
- Better UX

Pro Tip: Use Redis caching to reduce database load, improve response time, and deliver a smooth user experience even at high traffic.

Cache Smart, Deliver Fast!

DEPLOYMENT

Deploy Your Node.js App

Ape.js App To Production

Write Code. Deploy. Scale.

Deployment is the process of making your application available to users over the internet in a **production environment.**

DEPLOYMENT FLOW

1. Code Write & Test
2. Push To GitHub
3. Deploy To Platform
4. Live On Internet

PRODUCTION CHECKLIST

- Environment Variables
- Database Connection
- Security Best Practices
- Error Handling
- Logging
- CORS Configuration
- Rate Limiting
- Compression
- Health Check Endpoint
- Backup Strategy

EXAMPLE: DEPLOY NODE.JS APP

```
1  const express = require('express');
2  const app = express();
3
4  app.get('/', (req, res) => {
5    res.send('Hello from Production! 🚀');
6  });
7  const PORT = process.env.PORT || 3000;
8  app.listen(PORT,  () => {
9    console.log(`Server running on port ${PORT}`);
10 });
11
12
13 // Start
14 // node server.js
15 // Set PORT in environment variables
16 // Deploy to your favorite platform
```

ENVIRONMENT VARIABLES

```
PORT          = 3000
NODE_ENV      = production
DATABASE_URL  = mongodb://...
JWT_SECRET    = your_secret
CORS_ORIGIN   = https://yourdomain.com
REDIS_URL     = redis://...
EMAIL_USER    = user@example.com
EMAIL_PASS    = your_password
```

TIPS

- Never commit .env file to GitHub
- Use strong secrets and rotate regularly
- Use HTTPS for production
- Enable monitoring and logging
- Always test in staging first

POPULAR DEPLOYMENT PLATFORMS

RENDER

- Easy to use
- Built-in CI/CD
- Free Tier Available
- Auto SSL
- Great for Web Services

Best for: Full-stack apps, APIs, Databases

render.com

RAILWAY

- Developer Friendly
- GitHub Integration
- Environment Variables
- Databases & Services
- Usage-based Pricing

Best for: Modern Apps, Microservices

railway.app

VERCEL

- Blazing Fast
- Great for Frontend & APIs
- Serverless Functions
- Preview Deployments
- Global CDN

Best for: Frontend, Full-stack (Next.js), Serverless

vercel.com

AWS

- Highly Scalable
- Many Services
- EC2, ECS, Lambda
- S3, RDS, CloudFront
- Enterprise Grade

Best for: Large Scale Production Apps

aws.amazon.com

CUSTOM HOSTING

- Full Control
- Choose Your Stack
- Cost Effective
- Flexible Configuration
- Requires DevOps Knowledge

Best for: Advanced Users, Custom Requirements

Any VPS / Dedicated Server

DEPLOYMENT STEPS (GENERIC)

1	**Prepare Your App**	Clean code, set environment variables
2	**Push to GitHub**	Keep your code in a Git repository
3	**Connect Platform**	Connect your repo to deployment platform
4	**Configure Environment**	Add environment variables and settings
5	**Build & Deploy**	Platform builds and deploys automatically
6	**Go Live**	Your app is now live on the internet! 🎉

HOSTING OPTIONS COMPARISON

Platform	Ease of Use	Performance	Scalability	Free Tier	Best For
Render	★★★★★	★★★★☆	★★★★★	✓	Web Apps, APIs
Railway	★★★★★	★★★★☆	★★★★☆	✓	Modern Apps
Vercel	★★★★☆	★★★★★	★★★★☆	✓	Frontend, Next.js
AWS	★★★☆☆	★★★★★	★★★★★	Limited	Enterprise Apps
Custom VPS	★★☆☆☆	★★★☆☆	★★★★☆	Depends	Full Control

PRODUCTION ENVIRONMENT BEST PRACTICES

- Use PM2 or Process Manager
- Enable Compression
- Use HTTPS (SSL)
- Monitor Logs & Errors
- Backup Your Database
- Keep Dependencies Updated

Recommended: Use PM2 to keep your app running

```
# Install PM2
npm install -g pm2

# Start your app
pm2 start server.js --name my-app
pm2 save
pm2 startup
```

PM2

Keep Your App Alive Forever!

DEPLOYMENT ARCHITECTURE (EXAMPLE)

- High Availability (Always Online)
- Fast & Reliable Response
- Scalable On Demand
- Secure & Stable

Code Locally. Deploy Globally. Scale infinitely. 🚀

Choose Platform | Push Code | Deploy | Go Live | Happy Users

PM2

Process Manager

Production Process Manager for Node.js

PM2 is a production process manager for Node.js applications with a built-in load balancer. It keeps your app alive forever!

WHY USE PM2?

- **Auto Restart** — Keeps your app alive forever
- **Zero Downtime** — Restart application without downtime
- **Cluster Mode** — Utilize all CPU cores for maximum performance
- **High Performance** — Built-in load balancer for Node.js apps
- **Monitoring** — Real-time monitoring and performance metrics
- **Easy Management** — Simple CLI to manage your applications
- **Logs Management** — Advanced logging with search and filters
- **Developer Friendly** — Built for developers, loved by DevOps

EXAMPLE: ECOSYSTEM.CONFIG.JS

```
module.exports = {
  apps: [
    {
      name: 'my-app',
      script: 'app.js',
      instances: 'max',        // Cluster mode
      exec_mode: 'cluster',    // Cluster mode
      env: {
        NODE_ENV: 'production',
        PORT: 3000
      },
      log_file: './logs/combined.log',
      out_file: './logs/out.log',
      error_file: './logs/error.log',
      log_date_format: 'YYYY-MM-DD HH:mm Z',
      max_memory_restart: '500M',
      autorestart: true,
      watch: false
    }
};
```

QUICK INSTALL

```
$ npm install -g pm2
# or
$ yarn global add pm2
```

CHECK PM2 VERSION

```
$ pm2 --version
5.3.0
```

PM2 DAEMON

PM2 runs as a background daemon process to manage your applications.

```
$ pm2 startup
$ pm2 save
```

START

Start your application

```
$ pm2 start app.js
$ pm2 start ecosystem.config.js
```

Start with name

```
$ pm2 start app.js --name my-app
```

RESTART

Restart application

```
$ pm2 restart my-app
```

Restart all apps

```
$ pm2 restart all
```

STOP

Stop application

```
$ pm2 stop my-app
```

Stop all apps

```
$ pm2 stop all
```

LOGS

View application logs

```
$ pm2 logs my-app
```

Logs with options

```
$ pm2 logs my-app --lines 100
```

MONITOR

Monitor in real-time

```
$ pm2 monit
```

Show status

```
$ pm2 status
```

CLUSTER MODE

Run in cluster mode

```
$ pm2 start app.js -i max
```

Or with specific instances

```
$ pm2 start app.js -i 4
```

AUTO RESTART

Auto restart on failure

- Crash recovery
- Memory limit
- Watching files
- Always alive

CLUSTER MODE (MULTI-CORE)

PM2 Cluster Mode runs multiple instances of your app and load balances requests.

Utilizes all CPU cores for maximum performance!

COMMON COMMANDS CHEAT SHEET

Command	Description
`pm2 start app.js`	Start application
`pm2 start ecosystem.config.js`	Start with ecosystem config
`pm2 list`	List all running applications
`pm2 status`	Show status
`pm2 restart <name|id>`	Restart application
`pm2 stop <name|id>`	Stop application
`pm2 delete <name|id>`	Delete application
`pm2 logs <name|id>`	Show logs
`pm2 monit`	Monitor dashboard
`pm2 save`	Save current process list
`pm2 startup`	Generate startup script

MONITOR DASHBOARD (pm2 monit)

PM2 MONITOR [CPU: 12%] [MEM: 1.2G/7.8G] [LOAD: 0.45]

App Name	id	mode	pid	status	cpu%	mem	restart
my-app	0	fork	1234	online	2.3	45M	0
my-app	1	fork	1235	online	1.8	46M	0
my-app	2	fork	1236	online	2.1	44M	0
my-app	3	fork	1237	online	1.9	45M	0

AUTORESTART FEATURES

PM2 automatically restarts your app when:

- App crashes or exits unexpectedly
- Memory usage exceeds limit
- System reboot
- Uncaught exceptions
- Manual restart / reload

Keeps your app alive forever!

BEST PRACTICES

- Use ecosystem.config.js for complex apps
- Enable clustering for better performance
- Set memory limits (max_memory_restart)
- Monitor with pm2 moith regularly
- Use meaningful app names
- Check logs regularly

PM2 = Performance + Reliability + Simplicity

Write once, run forever!

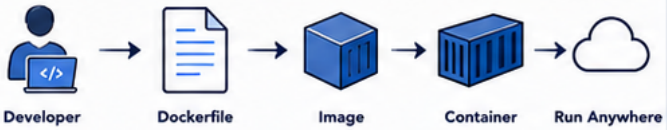
DOCKER

Build • Ship • Run • Anywhere

Docker helps you package your application with all its dependencies into a **container** so it runs consistently on any environment.

- Consistent Environment
- Fast Deployment
- Lightweight & Portable
- Secure Isolation
- DevOps Ready

HOW DOCKER WORKS



Developer	Dockerfile	Image	Container	Run Anywhere
Write your code	Define environment & dependencies	Built image (read-only)	Running instance of the image	Any OS, Any Cloud

Image = Read-only template with app & dependencies

Container = Runnable instance of an image

DOCKERFILE EXAMPLE

```
# Use official Node.js image
FROM node:18-alpine

# Set working directory
WORKDIR /app

# Copy package files
COPY package*.json ./
RUN npm install

# Copy application code
COPY . .

# Expose port
EXPOSE 3000

CMD ["node", "app.js"]
```

WHY USE DOCKER?

- **Consistency** — Run the same everywhere
- **Isolation** — Containers are isolated from each other
- **Efficiency** — Lightweight & uses fewer resources
- **Scalability** — Scale your apps easily
- **Portability** — Build once, run anywhere

1 DOCKERFILE

A Dockerfile is a text file that contains instructions to build a Docker image.

Dockerfile

```
FROM node:18-alpine
WORKDIR /app
COPY . .
RUN npm install
CMD ["node", "app.js"]
```

2 IMAGE

An image is a read-only template with everything needed to run an application.

Immutable
Can create many containers from the same image

3 BUILD

Build an image from a Dockerfile using the docker build command.

```
docker build -t my-app:1.0 .
```

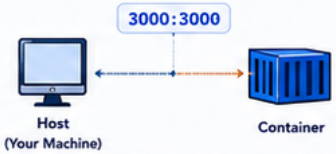
4 RUN

Run a container from an image using the docker run command.

```
docker run -d --name my-app
my-app:1.0
```

5 PORT MAPPING

Map container ports to host ports to access your app.




```
docker run -p 3000:3000
my-app:1.0
```

6 VOLUME

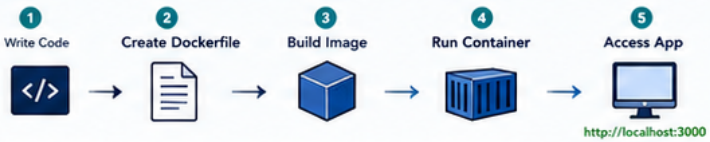
Volumes persist data outside the container lifecycle.

```
docker run -v mydata:/app/data
my-app:1.0
```

USEFUL DOCKER COMMANDS

Command	Description
`docker build -t <name> .`	Build an image from Dockerfile
`docker images`	List all images
`docker run <image>`	Run a container
`docker ps`	List running containers
`docker ps -a`	List all containers
`docker stop <container_id>`	Stop a running container
`docker rm <container_id>`	Remove a container
`docker rmi <image_id>`	Remove an image
`docker logs <container_id>`	View container logs

DOCKER WORKFLOW (EXAMPLE)




```
$ docker build -t my-app .
$ docker run -d -p 3000:3000 --name my-container my-app
$ docker ps
$ docker logs my-container
```

Result
Your application is running in a container!

TYPES OF VOLUMES

- **Named Volume** — Managed by Docker (Easy to backup & share)
- **Bind Mount** — Mount a host directory into the container
- **Anonymous Volume** — Created on the fly (Removed when container is removed)

PORT MAPPING EXAMPLE

Host Machine	Container
3000	3000
8080	80
5432	5432

Access your app at: http://localhost:3000

DOCKER IN A NUTSHELL

Build (Dockerfile) → (Image) → Run (Container) → Anywhere (Cloud/Server)

PRO TIP

Use `.dockerignore` to exclude unnecessary files and keep your images small and secure.

POPULAR USE CASES

- Web Apps
- Microservices
- CI/CD Pipelines
- Databases
- APIs

Docker makes your development faster, deployment easier, and applications portable!

TESTING

Quality Code. Confident Deploy.

Testing helps you find bugs early, improves code quality, and makes refactoring safe and easy.

- Reliable Code
- Early Bug Detection
- Confident Refactoring
- Faster Delivery
- Better Maintainability

POPULAR TESTING TOOLS

Jest

Delightful JavaScript Testing Framework by Facebook.

- ✓ Zero config
- ✓ Snapshot testing
- ✓ Great DX

Supertest

Super-agent driven library for testing HTTP APIs.

- ✓ Test REST APIs
- ✓ Chainable API
- ✓ Works with any framework

Mocha

Flexible JavaScript test framework for Node.js.

- ✓ Flexible & open
- ✓ Many reporters
- ✓ Async support

Chai

Assertion library that pairs well with any framework.

- ✓ Rich assertions
- ✓ BDD / TDD styles
- ✓ Extensible

TYPES OF TESTS

Unit Test

Test individual functions or modules in isolation. Fast, focused, reliable.

Integration Test

Test how multiple parts of the system work together. Slower, broader, realistic.

ASSERTIONS

Assertions are used to verify expected outcomes.

✓ expect(a). toBe(b)	Strict equality
✓ expect(a). toEqual(b)	Deep equality
✓ expect(a). toBeTruthy()	Truthiness
✓ expect(a). toContain(x)	Array / String contains
✗ expect(a). toThrow()	Function throws error

JEST EXAMPLE (Unit Test)

```
// sum.js
function sum(a, b) {
  return a + b;
}

// sum.test.js
const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});

test('handles negative numbers', () => {
  expect(sum(-1, -1)).toBe(-2);
});
```

Run: npx jest

SUPERTEST EXAMPLE (Integration Test)

```
// app.js (Express)
app.get('/api/users', (req, res) => {
  res.json([{ id: 1, name: 'Jeet' }]);
});

// users.test.js
const request = require('supertest');
const app = require('./app');

test('GET /api/users returns users', async () => {
  const res = await request(app)
    .get('/api/users')
    .expect(200);

  expect(res.body).toHaveLength(1);
  expect(res.body[0].name).toBe('Jeet');
});
```

Run: npx jest users.test.js

MOCHA EXAMPLE (BDD Style)

```
const assert = require('assert');
const { sum } = require('./sum');

describe('Sum Function', () => {
  it('should return correct sum', () => {
    assert.equal(sum(2, 3), 5);
  });

  it('should handle zeros', () => {
    assert.equal(sum(0, 0), 0);
  });
});
```

Run: npx mocha

CHAI EXAMPLE (Using Chai)

```
const chai = require('chai');
const expect = chai.expect;
const { sum } = require('./sum');

describe('Sum Function', () => {
  it('adds numbers correctly', () => {
    expect(sum(2, 3)).to.equal(5);
    expect(sum(1, -1)).to.equal(0);
  });

  it('returns a number', () => {
    expect(sum(5, 5)).to.be.a('number');
  });
});
```

Run: npx mocha

UNIT vs INTEGRATION

	Unit Test	Integration Test
Focus	Test small pieces (functions/modules)	Test how components work together
Speed	Very Fast	Slower
Dependencies	Mocked / Isolated	Real / Connected
Use Case	Business logic, utils, helpers	APIs, DB, Middleware, Routes
Example	sum(2,3) === 5	GET /api/users returns 200

TEST COVERAGE

Code coverage shows the percentage of code tested.

- Covered
- Partially Covered
- Not Covered

Statements Covered

```
Run coverage with Jest
$ npx jest --coverage
-------------|---------|----------|---------|---------|
File         | % Stmts | % Branch | % Funcs | % Lines |
-------------|---------|----------|---------|---------|
All files    |   85.71 |    80.00 |   90.00 |   86.20 |
sum.js       |  100.00 |   100.00 |  100.00 |  100.00 |
utils.js     |   66.66 |    50.00 |   66.66 |   66.66 |
-------------|---------|----------|---------|---------|
Test Suites: 2 passed, 2 total
Tests:       8 passed, 2 total
Snapshots:   2 passed, 2 total
Time:        1.345 s
```

TEST WORKFLOW

1. **Write Code** — Implement feature
2. **Write Tests** — Unit / Integration tests
3. **Run Tests** — Execute & verify behavior
4. **Fix Bugs** — Make tests pass
5. **Refactor** — Improve code with confidence

Red → Green → Refactor
Keep tests fast, isolated and reliable.

BEST PRACTICES

- ✓ Write tests before or with code (TDD / BDD)
- ✓ Keep tests independent and repeatable
- ✓ Use meaningful test names
- ✓ Test behavior, not implementation
- ✓ Maintain good test coverage
- ✓ Mock external services & DB in unit tests
- ✓ Keep tests fast

Pro Tip: Great tests = Confident Code + Happy Developers + Happy Users

Jest | Supertest | Mocha | Chai | Test More | Ship Better | Sleep Better

EXPRESS BEST PRACTICES

Write Better. Scalable. Secure. Maintainable.

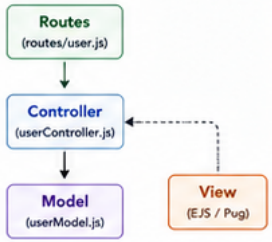
Why Best Practices?
Following best practices makes your Express.js applications secure, scalable, easy to maintain and production ready.

Scalable | Secure | Maintainable | Testable | Fast

1 MVC PATTERN

Organize your code using MVC for separation of concerns.

```
/project
├── /config
├── /controllers   → Handle request & response
├── /models        → Database schemas
├── /routes        → Define routes
├── /views         → Templating (if needed)
├── /middlewares   → Custom middlewares
├── /services      → Business logic
├── app.js         → Entry point
└── /utils         → Helpers & utilities
```



Benefit: Easy to manage, test and scale your application.

2 REUSABLE CODE

Create reusable modules, middlewares and utility functions.

```
// middlewares/auth.js
module.exports = function protect(req, res, next) {
  if (!req.user) {
    return res.status(401).json({
      message: 'Unauthorized'
    });
  }
  next();
};
```

Keep code DRY: Don't Repeat Yourself.

3 VALIDATION

Validate and sanitize user input.

```
const { body, validationResult } = require('express-validator');

// In route
router.post('/register', [
  body('email').isEmail().normalizeEmail(),
  body('password').isLength({ min: 6 })
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  // continue...
});
```

Validate early, sanitize always.

4 SECURITY

Secure your application and data.

- Use Helmet (set secure HTTP headers)
- Use CORS (allow only required origins)
- Rate Limiting (prevent brute force attacks)
- Sanitize Inputs (prevent NoSQL/SQL injection)
- Use HTTPS (never run on plain HTTP)
- Keep Dependencies Updated

```
const express = require('express');
const helmet = require('helmet');
const cors = require('cors');
const rateLimit = require('express-rate-limit');
const app = express();

app.use(helmet());
app.use(cors({ origin: 'https://yourdomain.com' }));

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 100
});
app.use(limiter);

app.use(express.json({ limit: '10kb' }));
```

5 ERROR HANDLING

Handle errors in a centralized way.

- Use try/catch in async functions
- Create custom error classes
- Use global error handling middleware
- Send meaningful error responses

```
// utils/AppError.js
class AppError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
    this.success = false;
  }
}
// middlewares/errorHandler.js
module.exports = (err, req, res, next) => {
  const status = err.statusCode || 500;
  res.status(status).json({
    success: false,
    message: err.message || 'Internal Server Error'
  });
};
```

6 ENVIRONMENT VARIABLES

Keep configuration in environment variables.

```
.env
PORT=5000
NODE_ENV=production
DB_URI=mongodb+srv://...
JWT_SECRET=your_secret
CLOUDINARY_CLOUD_NAME=xxx
```

Never commit .env file. Add it to .gitignore

```
// config/env.js
require('dotenv').config();

module.exports = {
  port: process.env.PORT || 5000,
  nodeEnv: process.env.NODE_ENV,
  dbUri: process.env.DB_URI,
  jwtSecret: process.env.JWT_SECRET
};
```

7 LOGGING

Use logging for better debugging and monitoring.

- Use a logger like Winston or Morgan
- Log requests, errors and important events
- Different levels (info, warn, error)
- Avoid logging sensitive data

```
const winston = require('winston');
const logger = winston.createLogger({
  level: 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.printf(info =>
      `${info.timestamp} [${info.level}]
       : ${info.message}`))
  },
  transports: [
    new winston.transports.Console(),
    new winston.transports.File({ filename: 'logs/err',
      log',  level: 'error' })
  ]
});
module.exports = logger;
```

8 CLEAN CODE

Write clean, consistent and readable code.

- Use meaningful names
- Keep functions small
- Use consistent formatting (Prettier / ESLint)
- Add comments where needed
- Remove unused code
- Follow Single Responsibility Principle

```
// Good
async function getUserById(id) {
  const user = await User.findById(id).select('-password');
  if (!user) throw new AppError('User not found', 404);
  return user;
}

// Bad
asnyc function f(id){
  const u = await User.findById(id)
  if(!u) return null;
  return u;
}
```

EXPRESS APP STRUCTURE EXAMPLE

```
/project
├── /config
│   └── db.js
├── /controllers
│   └── userController.js
├── /routes
│   └── userRoutes.js
├── /models
│   └── userModel.js
├── /middlewares
│   └── errorHandler.js
├── /utils
│   └── AppError.js
└── app.js
```

```
// app.js
require('dotenv').config();
const express = require('express');
const morgan = require('morgan');
const userRoutes = require('./routes/userRoutes');
const errorHandler = require('./middlewares/errorHandler');

const app = express();

app.use(morgan('dev'));
app.use(express.json());
app.use('/api/users', userRoutes);

app.use(errorHandler);

module.exports = app;
```

QUICK SUMMARY

Practice	Purpose
MVC Pattern	Organize code for scalability & maintainability
Reusable Code	Avoid duplication, use modules & middlewares
Validation	Ensure correct & safe user input
Security	Protect app & data from common attacks
Error Handling	Handle errors gracefully & consistently
Environment Variables	Keep configs secure & separate
Logging	Track events and debug issues
Clean Code	Improve readability and long-term maintainability

PRO TIP

Great code is not just about working, it's about being readable, secure and easy to maintain.

</>

Follow best practices today, build a better app tomorrow!

Remember: Best practices are not rules, but proven ways to build better applications.

</> Write Clean | Stay Secure | Think Scalable | Keep Improving | Happy Users

Better Code. Better Apps. Better Future.

COMMON EXPRESS ERRORS

Understand • Handle • Respond

express

Handling errors properly makes your API reliable, user-friendly and production ready.

BEST PRACTICES

- Always handle errors gracefully
- Send meaningful error messages
- Use proper status codes
- Don't expose sensitive information
- Log errors for debugging

GLOBAL ERROR HANDLER EXAMPLE

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  const statusCode = err.statusCode || 500;
  const message = err.message || 'Internal Server Error';
  res.status(statusCode).json({
    success: false,
    status: statusCode,
    message: message,
  });
});
```

ERROR RESPONSE FORMAT (EXAMPLE)

```
{
  "success": false,
  "status": 404,
  "error": "Not Found",
  "message": "Route not found",
  "path": "/api/unknown",
  "timestamp": "2025-05-25T12:00:00.000Z"
}
```

Tip: Be consistent with your error response format.

404 ERROR

Resource not found.

Example

```
res.status(404).json({
  success: false,
  message: 'Resource not found'
});
```

How to Fix: Check the URL and make sure the resource exists.

500 ERROR

Internal server error.

Example

```
res.status(500).json({
  success: false,
  message: 'Internal Server Error'
});
```

How to Fix: Check server logs and fix the issue in your code.

ROUTE NOT FOUND

The route does not exist.

Example

```
app.use((req, res) => {
  res.status(404).json({
    success: false,
    message: 'Route not found'
  });
});
```

How to Fix: Define the correct route or check the endpoint URL.

MIDDLEWARE ERROR

Error in custom or third-party middleware.

Example

```
app.use((req, res, next) => {
  throw new Error('Middleware failed');
});
```

How to Fix: Debug the middleware and handle errors properly.

DATABASE ERROR

Database connection or query failed.

Example

```
try {
  // DB Query
} catch (err) {
  res.status(500).json({
    success: false,
    message: 'Database error'
  });
}
```

How to Fix: Check DB connection, query syntax and credentials.

VALIDATION ERROR

Invalid or missing input data.

Example

```
if (!req.body.name) {
  return res.status(400).json({
    success: false,
    message: 'Name is required'
  });
}
```

How to Fix: Validate input using express-validator or custom checks.

JWT ERROR

Invalid, expired or missing token.

Example

```
jwt.verify(token, secret, (err, user) => {
  if (err) {
    return res.status(401).json({
      success: false,
      message: 'Invalid token'
    }
  }
});
```

How to Fix: Check token, secret key and token expiration.

CORS ERROR

Cross-Origin request blocked.

Example

```
No 'Access-Control-Allow-Origin'
header is present on the requested
resource.
```

How to Fix: Enable CORS in your app.

```
app.use(cors({ origin: '*' }));
```

Always handle errors. Never ignore them. | Detect | Understand | Fix | Test | Deploy | Build Reliable APIs!

express INTERVIEW QUESTIONS

Crack your Node.js + Express Interview

These questions cover core concepts of Express.js commonly asked in interviews.

Routing · Middleware · REST API · Security · Database · Best Practices

1 MIDDLEWARE

1. What is middleware in Express?
2. How does middleware work?
3. What are the types of middleware?
4. How do you create custom middleware?
5. How do you handle errors in middleware?
6. What is the order of middleware execution?

Example

```
// Custom Middleware
const logger = (req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next(); // Pass control to next middleware
};

app.use(logger);
app.get('/', (req, res) => res.send('Hi'));
```

Key Points

- app.use() for global middleware
- next() passes control
- Order matters
- Error middleware has 4 args

2 ROUTING

1. What is routing in Express?
2. How do you define a route?
3. What is route parameter?
4. What is query parameter?
5. How do you handle 404 routes?
6. Can we use regex in routes?

Example

```
app.get('/users/id',(req, res) => {
  const { id } = req.params;
  res.send('User ID: ${id}');
});

app.get('/search', (req, res) => {
  const { q } = req.query;
  res.send('Search: ${q}');
});
```

Common Status Codes

200 OK · 201 Created · 400 Bad Request · 404 Not Found · 500 Server Error

4 JWT (JSON WEB TOKEN)

1. What is JWT?
2. How does JWT authentication work?
3. How do you generate a token?
4. How do you verify a token?
5. What is the difference between JWT & Session?
6. How do you handle token expiration?

Example

```
// Generate Token
const token = jwt.sign({ id: user._id, role: user.role },
  process.env.JWT_SECRET, { expiresIn: '1h' });
// Verify Token Middleware
const verifyToken = (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];
  if (!token) return res.status(400).json({ message: 'No token' });
  jwt.verify(token, process.env.JWT_SECRET, (err, decoded) => {
   if (err) return res.status(401).json({ message: 'Invalid token' });
    req.user = deceded;
    next();
  },
};
```

Key Points

- Send token in Authorization header
- Keep JWT secret safe
- Use short expiry
- Verify on protected routes

5 COOKIES

1. What are cookies?
2. How do you set a cookie?
3. How do you read a cookie?
4. What are cookie options?
5. Difference between cookie & session?
6. How do you clear a cookie?

Example

```
// Set Cookie
res.cookie('token', 'abc123', {
  httpOnly: true,
  secure: true,
  sameSite: 'lax',
  maxAge: 24 * 60 * 60 * 1000 // 1 day
});

// Read Cookie
req.cookies.token;
```

Common Options

- **httpOnly** — Not accessible via JS
- **secure** — Only over HTTPS
- **sameSite** — CSRF protection
- **maxAge / expires** — Set expiration time

Use cookies to store small pieces of data on client side.

6 SESSIONS

1. What is a session?
2. How does session work in Express?
3. What is express-session?
4. How do you store session data?
5. How do you destroy a session?
6. Difference between session & JWT?

Example

```
const session = require('express-session');

app.use(session({
  secret: 'mySecretKey',
  resave: false,
  saveUninitialized: false,
  cookie: { maxAge: 1000 * 60 * 60 } // 1 hour
}));

req.session.user = { id: user._id,name: user.name };
req.session.destroy(() => {});
```

Session Flow

1. User Login
2. Create Session
3. Store on Server
4. Send Session ID (Cookie)
5. Use Session in Subsequent Requests

Session data is stored on server, session ID is stored in cookie.

7 EXPRESS ROUTER

1. What is Express Router?
2. Why do we use Express Router?
3. How do you create a router?
4. How do you mount a router?
5. Can we use middleware in router?
6. Difference between app.use() & router.use()?

Example

```
// user.routes.js
const router = require('express').Router();
router.get('/', getAllUsers);
router.post('/', createUser);
module.exports = router;

// app.js
const userRouter = require('./routes/user.routes');
app.use('/api/users', userRouter);
```

Benefits

- Modular Code
- Better Organization
- Easy to Maintain
- Scalable Structure

Use Router to keep routes modular and clean.

8 MVC PATTERN

1. What is MVC pattern?
2. How does MVC improve code structure?
3. What are Models in MVC?
4. What are Views in MVC?
5. What are Controllers in MVC?
6. How do you implement MVC in Express?

Folder Structure Example

```
project/
├── controlles/      → Handle logic
│   └── user.controller.js
├── models/          → Database models
│   └── User.js
├── routes/          → Route definitions
│   └── user.routes.js
├── views/           → Templates (EJS/Pug)
│   └── index.ejs
└── app.js           → Entry point
```

Key Points

- Separation of Concerns
- Easy Testing
- Reusability
- Better Maintainability

Pro Tip: Practice building small projects like Blog App, Todo App, Auth System.

Understand Concepts · Write Code · Test APIs · Handle Errors · Stay Updated

Practice. Code. Debug. Succeed!

USEFUL npm PACKAGES

Powerful packages every Node.js developer should know

Save Time — Avoid reinventing the wheel

Easy to Use — Install and integrate in seconds

Battle Tested — Used by millions of developers

Improve Quality — Write better, secure and scalable apps

1 Express

Fast, unopinionated, minimalist web framework for Node.js.

```
npm install express
```

Example

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello Express!');
});

app.listen(3000, () => console.log('Server running'));
```

Use Cases

- Build REST APIs
- Web Applications
- Microservices

2 Mongoose

Elegant MongoDB object modeling for Node.js.

```
npm install mongoose
```

Example

```
const mongoose = require('mongoose');
mongoose.conrect('mongodb://localhost:27017/mydb');

const userSchema = new mongoose.Schema({
  name: String,
  email: String
});
const User = mongoose.model('User', userSchema);
```

Use Cases

- MongoDB Modeling
- Schema Validation
- Relationships

3 dotenv

Loads environment variables from a `.env` file into `process.env`.

```
npm install dotenv
```

Example

```
require('dotenv').config();

const port = process.env.PORT;
const dbUrl = process.env.MONGO_URI;
console.log('Server running on ${port}');
```

Use Cases

- Environment Variables
- Configuration
- Secrets

4 bcrypt

Library to help you hash passwords securely.

```
npm install bcrypt
```

Example

```
const bcrypt = require('bcrypt');

// Hash Password
const hash = await bcrypt.hash('mypassword', 10);

// Compare Password
const match = await bcrypt.compare('mypassword', hash);
```

Use Cases

- Password Hashing
- User Authentication
- Security

5 jsronnokokten

JSON Web Token implementation for authentication.

```
npm install jsonwektoben
```

Example

```
const jwt = require('jsormokteon');

// Sign Token
const token = jwt.sign({ id: user.id }, 'secretKey', {
  expiresIn: '1h'
});

// Verify Token
const decoded = jwt.verify(token, 'secretKey');
```

Use Cases

- Authentication
- Authorization
- Secure APIs

6 Multer

Middleware for handling multipart/form-data (file uploads).

```
npm install multer
```

Example

```
const multer = require('multer');
const upload = multer({ dest: 'uploads/' });

app.post('/upload', upload.single('image'), (req, res) => {
  res.send('File uploaded successfully');
});
```

Use Cases

- File Uploads
- Image Uploads
- Form Data

7 morgan

HTTP request logger middleware for Node.js.

```
npm install morgan
```

Example

```
const morgan = require('morgan');
app.use(morgan('dev'));

// Custom format
app.use(morgan(':method :url
      :status :res[content-length] - :response-time ms'));
```

Use Cases

- Logging Requests
- Debugging
- Monitoring

8 cors

Enable Cross-Origin Resource Sharing in your Express app.

```
npm install cors
```

Example

```
const cors = require('cors');
app.use(cors());

// With options
app.use(cors({
  origin: 'https://example.com',
  methods: ['GET', 'POST', 'PUT', 'DELETE'],
  credentials: true
}));
```

Use Cases

- Enable CORS
- Secure APIs
- Cross-Origin Requests

Pro Tip: Always check the official documentation for latest features, options and best practices.

Docs https://docs.npmjs.com/

Search https://www.npmjs.com/

Install npm install <package-name>

Keep your packages updated for security and performance.

EXPRESS DEVELOPER TOOLS

express

Essential tools that every Express.js developer should know

Boost Productivity · Debug Easily · Test APIs · Build Better Apps

1 VS Code

Lightweight and powerful code editor.

Key Features

- IntelliSense & auto-completion
- Debugging support
- Extensions marketplace
- Git integration

```
// app.js
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello Express!');
});

app.listen(3000);
```

Use For: Writing, debugging and running Express apps

2 Postman

API platform for building, testing and documenting APIs.

Key Features

- Send HTTP requests
- Save & organize requests
- Environment variables
- Test scripts

GET http://localhost:3000/api/users

Params Auth Headers (7) Body Tests

Body Cookies Headers (7) Test Results

Pretty Raw Preview JSON

```
{
  "users": [
    { "id": 1, "name": "Jeet", "role": "dev" }
  ]
}
```

Use For: Testing REST APIs and automating API workflows

3 Thunder Client

Fast and lightweight API client extension for VS Code.

Key Features

- Send HTTP requests
- Environment variables
- Collections
- Code generation

GET http://localhost:3000/api/users Send

Query Headers Auth Body

Status: 200 OK Time: 45 ms Size: 215 B

```
{
  "users": [
    { "id": 1, "name": "Jeet", "role": "dev" }
  ]
}
```

Use For: API testing directly inside VS Code

4 MongoDB Compass

GUI tool to explore and manage MongoDB databases.

Key Features

- Visualize data
- Run queries
- Analyze performance
- Manage indexes

localhost:27017 > mydb > users

Documents Aggregations Schema Indexes

Filter ( )

```
_id: 6658c3f8f1a2b3c4d5e6f001
name: "Jeet"
email: "jeet@example.com"
role: "admin"

_id: 6658c3f8f1a2b3c4d5e6f002
name: "Alsh"
email: "alsh@example.com"
role: "user"
```

Use For: Database management and data exploration

5 Git

Distributed version control system.

Key Features

- Track changes
- Branching & merging
- Collaboration
- Rollback changes

Common Commands

```
git init
git add .
git commit -m "Initial commit"
git status
git push origin main
```

Use For: Version control and team collaboration

6 Node.js

JavaScript runtime built on Chrome's V8 engine.

Key Features

- Run JavaScript on server
- Event-driven, non-blocking I/O
- NPM ecosystem
- High performance

```
$ node -v
v20.11.1

$ node app.js
Server running on port 3000
```

Use For: Running Express applications

7 npm

Node package manager for JavaScript.

Key Features

- Install packages
- Manage dependencies
- Run scripts
- Publish packages

Common Commands

```
npm init -y
npm install express
npm install --save-dev nodemon
npm outdated
npm update
```

Use For: Managing packages and project dependencies

8 Chrome DevTools

Built-in browser tools for debugging and performance.

Key Features

- Inspect elements
- Console & logs
- Network requests
- Performance profiling

Elements Console Sources Network

Filter

Filter All Fetch/XHR JS CSS Img Media

Name	Status	Type	Time
users	200	fetch	45 ms
app.js	200	script	12 ms
styles.css	200	css	8 ms
logo.png	200	png	15 ms

Use For: Debugging, performance and network analysis

Pro Tip: Combine these tools for a faster and smoother development experience!

Typical Express.js Development Flow

Write Code (VS Code) → Test API (Postman/Thunder) → Check DB (Compass) → Commit & Push (Git) → Deploy App (Node.js) → Debug & Analyze (DevTools)

express DEVELOPMENT WORKFLOW

Step-by-step workflow to build, test and deploy robust Express.js applications

Structured Process · Faster Development · Fewer Bugs · Production Ready · Better Monitoring

1 CREATE PROJECT

- Create project folder
- Initialize npm
- Create basic structure

Example

```
mkdir my-express-app
cd my-express-app
npm init -y
```

```
my-express-app/
├── src/
│   ├── routes/
│   ├── controllers/
│   ├── models/
│   └── app.js
├── .env
├── package.json
└── .gitignore
```

Deliverable

Project skeleton is ready

2 INSTALL PACKAGES

- Install Express & core deps
- Install dev dependencies
- Save in package.json

Example

```
npm install express
cors dotenv mongoose

npm install -D nodemon
```

Key Packages

- express - Web framework
- mongoose - MongoDB ODM
- cors - Cross-Origin Support
- dotenv - Env management
- nodemon - Auto restart

Deliverable

All required packages installed

3 BUILD ROUTES

- Define API routes
- Create controllers
- Set up middleware

Example (src/routes/user.js)

```
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
  res.json({ message: 'Users API' });
});

module.exports = router;
```

Deliverable

API routes with controllers

4 CONNECT DATABASE

- Configure database
- Create models
- Test connection

Example (src/config/db.js)

```
const mongoose = require('mongoose');

mongoose.connect(process.env.MONGO_URI)
  .then(() => console.log('DB Connected'))
  .catch((err) => console.error(err));
```

.env Example

```
MONGO_URI=mongodb://127.0.0.1:27017/mydb
```

Deliverable

Database connected and models ready

5 TEST API

- Test endpoints
- Validate request/response
- Use tools like Postman or Thunder Client

Example (GET /api/users)

```
GET  http://localhost:3000/api/users

Params  Auth  Headers (7)  Body

Status: 200 OK  Time: 28 ms

{
  "users": [
    { "id": 1, "name": "Jeet", "age":25
  }
}
```

Deliverable

APIs tested and working

6 DEBUG

- Use logs & breakpoints
- Check errors & exceptions
- Fix and optimize code

Example

```
// Add logging
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});

// Error handler
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ message: 'Server Error' });
});
```

Deliverable

Bug-free and stable application

7 DEPLOY

- Push code to Git
- Deploy to hosting platform
- Set environment variables

Example (Render / Railway / Vercel)

```
git add .
git commit -m "Initial commit"
git push origin main
```

Render · Railway · Vercel · Heroku

.env on Server

Set all environment variables securely

Deliverable

Application deployed and live

8 MONITOR

- Monitor performance
- Track logs & errors
- Set up alerts

Tools

- Render Logs
- UptimeRobot
- Sentry
- PM2
- Winston (Logging)

Example (PM2)

```
npm install -g pm2
pm2 start src/app.js --name api
pm2 monit
```

Deliverable

Application monitored and healthy

Recommended Project Structure

```
my-express-app/
├── src/
│   ├── config/          # DB, env config
│   ├── controllers/     # Route controllers
│   ├── models/          # Database models
│   ├── routes/          # API routes
│   ├── middleware       # Custom middleware
│   # Helper functions
├── .env
├── package.json
├── .gitignore
├── .env
├── .gitignore
└── package.json
```

Workflow Tips

- Use environment variables for sensitive data
- Follow MVC pattern for clean code
- Write reusable middleware
- Validate requests and handle errors
- Use Git for version control
- Test before deploying

Typical Commands

```
# Start development server
npm run dev
# Run in production
npm start
# Run tests (if available)
npm test
```

Common Tools Used

VS Code · Postman · Thunder Client · MongoDB Compass · Git · Node.js · npm · Chrome DevTools

Pro Tip: Follow this workflow for every project to save time, reduce bugs and build production-ready APIs!

Plan → Code → Test → Debug → Deploy → Monitor → Improve

Build. Test. Deploy. Repeat.
Ship better code, faster!

EXPRESS PROJECT CHECKLIST

Complete checklist to build a production-ready Express.js application

GOAL
Build a secure, scalable, well-tested and well-documented Express.js API.

Reliable API | Secure By Default | Easy to Maintain | Scalable Architecture

1 REST API

Design and build clean and consistent REST APIs.

- [x] Follow RESTful conventions
- [x] Use meaningful endpoints
- [x] Use proper HTTP methods (GET, POST, PUT, DELETE)
- [x] Consistent response format
- [x] Implement pagination, filtering, sorting
- [x] Handle query parameters

Example Route

```
router.get('/api/posts', getPosts);
router.post('/api/posts', createPost);
router.get('/api/posts/:id', getPostById);
router.put('/api/posts/:id', updatePost);
router.delete('/api/posts/:id', deletePost);
```

Key Deliverable: Fully functional RESTful API

2 AUTHENTICATION

Implement user authentication and authorization.

- [x] User registration & login
- [x] Password hashing (bcrypt)
- [x] JWT generation and verification
- [x] Protect routes with middleware
- [x] Role-based access control (RBAC)
- [x] Refresh token (optional)

Example (Login)

```
const token = jwt.sign(
  { id: user._id, role: user.role },
  process.env.JWT_SECRET,
  { expiresIn: '1d' }
);
```

Key Deliverable: Secure authentication system

3 VALIDATION

Validate and sanitize all inputs to prevent bad data.

- [x] Validate request body, params, query
- [x] Use express-validator / Joi
- [x] Sanitize user inputs
- [x] Return clear validation errors
- [x] Validate ObjectId, email, etc.

Example (express-validator)

```
body('email').isEmail().withMessage('Invalid email'),

body('password').isLength({ min: 6 })
  .withMessage('Password too short')
```

Key Deliverable: Clean and valid input data

4 DATABASE

Design schema and interact with the database efficiently.

- [x] Use Mongoose ODM
- [x] Define proper schemas & models
- [x] Validate schema at DB level
- [x] Use indexes for performance
- [x] Handle relationships (refs, populate)
- [x] Write reusable DB queries

Example (Mongoose Schema)

```
const postSchema = new mongoose.Schema({
  title: { type: String, required: true },
  content: String,
  author: { type: mongoose.Schema.Types.ObjectId,
            ref: 'User' },
}, { timestamps: true });
```

Key Deliverable: Optimized and reliable database layer

5 SECURITY

Secure your API and protect against common threats.

- [x] Use HTTPS
- [x] Use Helmet.js for security headers
- [x] Rate limiting (express-rate-limit)
- [x] Prevent NoSQL injection
- [x] Sanitize inputs (xss-clean)
- [x] Enable CORS (with restrictions)
- [x] Hide sensitive data in responses

Example (Helmet)

```
const helmet = require('helmet');
app.use(helmet());
```

Key Deliverable: Hardened and secure application

6 TESTING

Write tests to ensure your API works as expected.

- [x] Write unit tests (controllers, utils)
- [x] Write integration tests (routes)
- [x] Use Jest / Mocha + Chai
- [x] Use Supertest for API testing
- [x] Maintain good test coverage
- [x] Run tests in CI pipeline

Example (Supertest + Jest)

```
test('GET /api/posts should return posts', async () => {
  const res = await request(app).get('/api/posts');
  expect(res.statusCode).toBe(200);
  expect(Array.isArray(res.body)).toBe(true);
});
```

Key Deliverable: Tested and bug-free API

7 DEPLOYMENT

Deploy your API to production environment.

- [x] Use environment variables
- [x] Build for production (NODE_ENV=production)
- [x] Use process manager (PM2)
- [x] Deploy to platform (Render, Railway, AWS, etc.)
- [x] Configure domain & SSL
- [x] Setup CI/CD pipeline (optional)
- [x] Backup strategy

Example (PM2)

```
pm2 start app.js --name my-express-api
pm2 save
pm2 startup
```

Key Deliverable: Live and accessible API

8 DOCUMENTATION

Document your API for users and future developers.

- [x] Create API documentation (Swagger / OpenAPI)
- [x] Document endpoints, params, responses
- [x] Provide request/response examples
- [x] Add authentication details
- [x] Keep README updated
- [x] Add contribution guide (optional)

Example (Swagger)

```
/**
 * @swagger
 * /api/posts:
 *   get:
 *     summary: Get all posts
 *     responses:
 *       200:
 *         description: List of posts
 */
```

Key Deliverable: Well-documented API

PROJECT STRUCTURE EXAMPLE

```
my-express-app/
├── src/
│   ├── config/
│   ├── controllers/
│   ├── models/
│   ├── routes/
│   ├── middleware/
│   ├── utils/
│   └── app.js
├── tests/
├── .env
├── .env
├── package.json
└── README.md
```

Pro Tip
Follow best practices, write clean code and keep improving!

TECH STACK (Example)

Node.js | Express.js | MongoDB | JWT | Jest | Supertest

QUALITY CHECK

- [x] Code follows best practices
- [x] No ESLint errors
- [x] > 80% test coverage
- [x] All API endpoints tested
- [x] Security checks passed

PROJECT READY!

Plan Define requirements → **Code** Build your API → **Test** Test thoroughly → **Debug** Fix issues → **Deploy** Go live → **Monitor** Monitor & improve

> Good code + Good tests + Good docs = Successful Project

EXPRESS FOLDER STRUCTURE

A clean, scalable and maintainable structure for Express.js projects

Organized — Easy to navigate
Scalable — Grows with your app
Maintainable — Easy to maintain
Production — Production ready

PROJECT TREE

```
my-express-app/
  src/
    config/
    routes/
    controllers/
    models/
    middleware/
    services/
    utils/
    app.js
    server.js
  .env
  .gitignore
  package.json
  README.md
```

1 src/

Root directory for all source code.

`src/`

Example Files

- app.js
- server.js

2 routes/

All route definitions (API endpoints).

`src/routes/`

Example Files

- user.routes.js
- auth.routes.js
- post.routes.js

3 controllers/

Request handlers and business logic.

`src/controllers/`

Example Files

- user.controller.js
- auth.controller.js
- post.controller.js

4 models/

Database models and schemas (Mongoose).

`src/models/`

Example Files

- user.model.js
- post.model.js
- comment.model.js

5 middleware/

Custom and third-party middleware functions.

`src/middleware/`

Example Files

- auth.middleware.js
- error.middleware.js
- validate.middleware.js

6 services/

Reusable business logic and external services.

`src/services/`

Example Files

- user.service.js
- email.service.js
- token.service.js

7 utils/

Utility/helper functions used across the app.

`src/utils/`

Example Files

- catchAsync.js
- appError.js
- response.js
- generateToken.js

8 config/

Configuration files and application settings.

`src/config/`

Example Files

- db.js
- env.js
- cloudinary.js

WHY THIS STRUCTURE?

- Separation of concerns
- Easy to manage & scale
- Better for team collaboration
- Production ready

IMPORT EXAMPLE (app.js)

```
const express = require('express');
const app = express();

const userRoutes = require('./routes/user.routes');
app.use('/api/users', userRoutes);

module.exports = app;
```

BENEFITS

- Easier navigation
- Better code reusability
- Easier testing
- Team friendly
- Ready for Production

BEST PRACTICES

- Keep controllers thin, put logic in services
- Use middleware for cross-cutting concerns
- Use meaningful file & folder names
- Handle errors in a centralized way
- Follow consistent coding style

WORKFLOW

Request → Route → Middleware → Controller → Service → Model (DB) → Response

Happy Coding!

EXPRESS REQUEST FLOW

Step-by-step journey of a request in an Express.js application

Fast — Lightweight & Minimal

Flexible — Highly Customizable

Powerful — Robust Features

Secure — Built-in Best Practices

1 CLIENT

Client (e.g. Browser, Mobile App, Postman) sends an HTTP request to the server.

2 REQUEST

HTTP Request is sent with method, headers, query, params and body (if any).

Example:

```
POST /api/users
Content-Type: application/json
Authorization: Bearer <token>
```

3 MIDDLEWARE

Request passes through **middleware functions.** (e.g. auth, logger, CORS, body-parser, etc.)

Example Middleware Stack:

```
app.use(cors());
app.use(express.json());
app.use(logger);
app.use(authMiddleware);
```

4 ROUTE

Express matches the request with the correct route based on method and path.

Example Route:

```
router.post('/users',
userController.createUser);
```

5 CONTROLLER

Route calls the controller function to handle business logic and process data.

Example Controller:

```
exports.createUser = async
(req, res) => {
  // logic here
};
```

6 DATABASE

Controller interacts with the database (CRUD operations) and gets the result.

Example (MongoDB):

```
const user = await User.
create(req.body);
// or
const users = await User.
find();
```

7 RESPONSE

Controller sends back a response (success or error) to the client using res object.

Example Response:

```
res.status(201).json({
  success: true,
  data: user
});
```

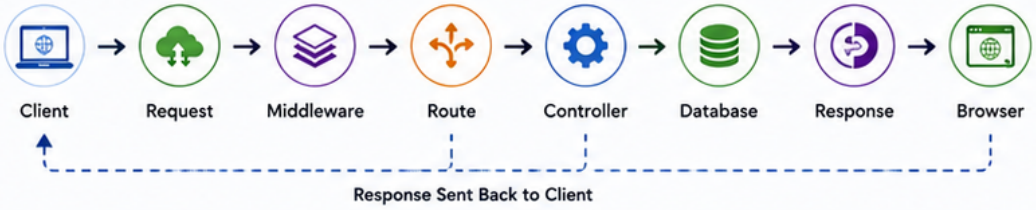
8 BROWSER

Client receives the response and displays data to the user.

Example JSON:

```
{
  "success": true,
  "data": { "id": 1,
  "name": "Jeet" }
}
```

VISUAL FLOW DIAGRAM



SIMPLE EXPRESS EXAMPLE

```
app.post('/api/users', authMiddleware, async (req, res) => {
  try {
    const user = await User.create(req.body);
    res.status(201).json({ success: true, data: user });
  } catch (error) {
    res.status(500).json({ success: false, message: error.message });
  }
});
```

WHY THIS FLOW MATTERS?

- Keeps code organized and maintainable
- Separation of concerns
- Easier to test and debug
- Better scalability
- Improved security

PRO TIP — Use middleware smartly, keep controllers clean, and handle errors gracefully for a robust API.

COMMON MISTAKE — Putting all logic in routes instead of using controllers makes code hard to manage and test.

BEST PRACTICE — Follow MVC pattern, validate input, handle errors, and return meaningful responses.

express JS

Fast • Minimal • Powerful

EXPRESS API LIFECYCLE

Journey of a Request in an Express.js Application

Fast — High Performance
Modular — Flexible & Scalable
Secure — Built-in Best Practices
Reliable — Stable & Production Ready

1 CLIENT REQUEST

Client (Browser, Mobile App, Postman, etc.) sends an HTTP request to the server.

Example

```
POST /api/users/login
Content-Type: application/json
{
  "email": "jeet@example.com",
  "password": "123456"
}
```

2 ROUTE MATCH

Express matches the incoming request with the defined route.

Example

```
app.post('/api/users/login',
  authMiddleware,
  validateLogin,
  loginController
);
```

3 MIDDLEWARE

Request passes through middleware functions (e.g., logger, auth, CORS, rate limiter).

Example

```
app.use(logger);
app.use(cors());
app.use(authMiddleware);
app.use(rateLimiter);
```

4 VALIDATION

Validate & sanitize input data to ensure it is safe and correct.

Example (Joi)

```
const schema = Joi.object({
  email: Joi.string().email()
    .required();
  password: Joi.string()
    .min(6).required()
});
```

5 CONTROLLER

Controller handles the request, executes business logic and prepares data.

Example

```
exports.login = async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });
  // check password, generate token
  res.locals.user = user;
};
```

6 DATABASE

Controller interacts with the database to fetch or store data.

Example (MongoDB)

```
const user = await User
  .findOne({ email })
  .select('+password');
```

7 JSON RESPONSE

Server sends back a JSON response to the client.

Example

```
res.json({
  success: true,
  message: 'Login successful',
  data: { token, user }
});
```

8 STATUS CODE

Response includes an appropriate HTTP status code.

Example

```
200 OK    - Success
201 Created - Resource
400 Bad Request - Invalid
401 Unauthoriced - Auth Fail
500 Server Error - Server Issue
```

LIFECYCLE FLOW DIAGRAM

COMPLETE EXAMPLE (EXPRESS)

```
app.post('/api/users/login', authMiddleware, validateLogin, async (req, res, next) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ email }).select('+password');
    if (!user || !(await user.comparePassword(password))) {
      return res.status(401).json({ success: false, message: 'Invalid credentials' });
    }
    const token = generateToken(user._id);
    return res.status(200).json({ success: true, data: { token, user } });
  } catch (err) {
    next(err); // error middleware
});
```

COMMON STATUS CODES

Code	Name	When to Use
200	OK	Request successful
201	Created	Resource created successfully
400	Bad Request	Invalid input or data
401	Unauthorized	Authentication failed
403	Forbidden	Access denied
404	Not Found	Route or resource not found
500	Internal Server Error	Server-side error occurred

KEY TAKEAWAY

- Every request goes through a defined lifecycle.
- Middleware and validation protect your app.
- Controllers keep business logic clean.
- Always return proper status codes.
- JSON response makes APIs easy to consume.

BEST PRACTICES

- Use middleware for cross-cutting concerns.
- Validate all incoming data.
- Keep controllers clean and focused.
- Handle errors using error middleware.
- Always return consistent responses.

PRO TIP

Use a consistent API response format in your app.

```
{
  "success": true,
  "message": "Operation successful",
  "data": {},
  "status": 200
}
```

TOOLS YOU CAN USE

Postman · VS Code · Morgan · Joi · Mongoose

Remember: A well-structured API lifecycle = Fast, Secure, and Scalable Application

Request → Route → Middleware → Validation → Controller → Database → Response → Status Code

EXPRESS PROJECT ARCHITECTURE

Scalable, Secure & Maintainable Node.js + Express.js + MongoDB Application

MODULAR Separation of Concerns | **SCALABLE** Easy to Scale & Maintain | **SECURE** Built-in Security Best Practices | **PERFORMANCE** Optimized for High Performance

1. FRONTEND

React, Vue.js, HTML/CSS/JS or Mobile App

Sends HTTP Requests
Receives JSON Responses

HTTP Request →
← JSON Response

2. EXPRESS API (SERVER)

- RESTful API
- JSON Data Format
- Stateless
- Fast & Minimal

server.js

```
server.js
const express = require('express');
const app = express();

app.use(express.json());
app.use('/api', require('./routes'));

app.listen(5000, () =>
  consele.log('Server running on 5000'));
```

3. MIDDLEWARE

- Authentication (JWT)
- Logger (morgan)
- CORS
- Rate Limiting
- Error Handler
- Body Parser

Example (app.js)

```
app.use(morgan('dev'));
app.use(cors());
app.use(helmet());
app.use(authMiddleware);
app.use(errorHandler);
```

4. CONTROLLERS

Handle requests & return responses

- Auth Controller
- User Controller
- Post Controller
- Comment Controller

Example (user.controller.js)

```
exports.getUsers = async (req, res) => {
  const users = await User.find();
  res.status(200).json(users);
};
```

5. MODELS

Define schema & interact with DB

- User Model
- Post Model
- Comment Model

Example (user.model.js)

```
const userSchema = new Schema({
  name: String,
  email: String,
  password: String
});
module.exports = mongoose.model('User',
  userSchema);
```

6. MONGODB

Collections

- users
- posts
- comments
- ...

Data Storage

7. AUTHENTICATION (JWT)

User → Login Request (email, password) → Express API → Validate User → Database
JWT Token

- User logs in with credentials
- Server validates user
- Server generates JWT Token
- Token sent to client
- Client includes token in headers for protected routes

Protected Route Example

```
app.get('/api/profile', authMiddleware, (req, res) => {
  res.json({ message: 'Protected Data', user: req.user });
});
```

ARCHITECTURE FLOW

Frontend (Client) → Express API (Server) → Middleware (Pipeline) → Controllers (Logic) → Models (Schema) → MongoDB (Database)

REQUEST FLOW

Client → Express API → Middleware → Controller → Model → MongoDB
MongoDB → Model → Controller → Response → Client

PROJECT STRUCTURE

```
my-express-app/
├── src/
│   ├── config/
│   ├── middlewares/
│   ├── controllers/
│   ├── models/
│   ├── routes/
│   ├── utils/
│   └── app.js
├── .env
├── server.js
└── package.json
```

config/	- DB, env, app config
middlewares/	- Custom middlewares
controllers/	- Request handlers
models/	- Mongoose models
routes/	- API routes
utils/	- Helper functions
app.js	- Express app setup
server.js	- Start the server

WHY THIS ARCHITECTURE?

- Separation of concerns
- Easy to maintain
- Scalable & extensible
- Secure (Auth + Middleware)
- Clean & modular codebase
- Industry best practices

TECH STACK

Node.js, express.js, MongoDB, Mongoose, JWT, npm

BEST PRACTICES

- Use ENV variables
- Secure passwords
- Validate input
- Handle errors
- Use HTTPS
- Keep dependencies updated

TOOLS

VS Code, Postman, MongoDB Compass, Git & GitHub

EXPRESS.JS QUICK REFERENCE

A Quick Cheat Sheet for Building Fast & Scalable Node.js Web Applications with Express

Fast Minimal Overhead | **Flexible** Unopinionated | **Scalable** Built for Growth | **Robust** Production Ready

1 ROUTING

Define routes to handle different HTTP requests.

```
GET     app.get('/', (req, res) => {})
POST    app.post('/api/data', (req, res) => {})
PUT     app.put('/api/data/:id', (req, res) => {})
DELETE  app.delete('/api/data/:id', (req, res) => {})
ALL     app.all('*', (req, res) => {})
```

```
// Simple Route Example
app.get('/hello', (req, res) => {
  res.send('Hello from Express! 🎉');
});
```

2 MIDDLEWARE

Functions that run during the request-response cycle.

- Execute code
- Modify request/response
- End the request-response cycle
- Call the next middleware

```
// Custom Middleware
const logger = (req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next(); // pass control to next middleware
};

app.use(logger);
app.use(express.json());  // Built-in middleware
```

3 REST API

Build RESTful APIs with standard HTTP methods.

Method	Purpose	Example Endpoint
GET	Retrieve data	/api/users
POST	Create data	/api/users
GET	Retrieve by ID	/api/users/:id
PUT	Update by ID	/api/users/:id
DELETE	Delete by ID	/api/users/:id

```
// Example: Get Users
app.get('/api/users', async (req, res) => {
  const users = await User.find();
  res.status(200).json(users);
});
```

4 EXPRESS ROUTER

Create modular, mountable route handlers.

```
// routes/user.routes.js
const express = require('express');
const router = express.Router();

router.get('/', getAllUsers);
router.post('/', createUser);

module.exports = router;
```

```
// app.js
const userRoutes = require('./routes/user.routes');
app.use('/api/users', userRoutes);
```

5 MONGODB

Connect and interact with MongoDB using Mongoose.

```
// Connect
const mongoose = require('mongoose');
await mongoose.connect(process.env.MONGO_URI);
console.log('MongoDB Connected ✅');
```

```
// Schema & Model
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  createdAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model('User', userSchema);
```

Common Operations

- User.find()
- User.findById(id)
- User.create(data)
- User.updateOne()
- User.deleteOne()

Tip

Use .lean() for better performance on read operations.

6 JWT (AUTHENTICATION)

Secure your API with JSON Web Tokens.

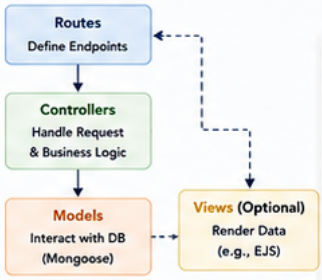
```
// Generate Token (Login)
const jwt = require('jsonwebtoken');
const token = jwt.sign(
  { id: user._id, email: user.email },
  process.env.JWT_SECRET,
  { expiresIn: '7d' }
);
res.json({ token });
```

```
// Protect Routes
const auth = (req, res, next) => {
const token = req.headers.authorization?.split(' ')[1];
if (!token) return res.status(401).json({ msg: 'No token' });
try {
  req.user = jwt.verify(token, process.env.JWT_SECRET);
  next();
} catch (err) {
  res.status(403).json({ msg: 'Invalid token' });
}
};

app.use('/api/protected', auth, protectedRoutes);
```

7 MVC (ARCHITECTURE)

Organize your project using MVC pattern.



Folder Structure

```
src/
├─ config/
├─ routes/
│  └─ user.routes.js
├─ controllers/
│  └─ user.controller.js
├─ models/
│  └─ user.model.js
├─ middleware/
│  └─ auth.js
├─ utils/
├─ app.js
└─ server.js
```

Benefits: Clean Code | Easy Maintenance | Scalable

8 DEPLOYMENT

Deploy your Express app to the world.

Popular Platforms

- Render
- Railway
- Cyclic
- Heroku
- Vercel (Serverless)

Deployment Steps

1. Push code to GitHub
2. Build the app
3. Set environment variables
4. Deploy
5. Monitor & Scale

```
// Start Command
npm start  // for production
nodemon server.js   // for development
```

QUICK COMMANDS

```
npm init -y                  # Initialize project
npm install express          # Install Express
npm install mongoose         # Install Mongoose
npm install dotenv           # Environment variables
npm install jsonwebtoken     # JWT
npm install cors             # CORS support
nodemon server.js            # Run in dev mode
```

BEST PRACTICES

- Use environment variables
- Validate user input
- Use middleware for common tasks
- Handle errors properly
- Keep controllers clean & focused
- Use async/await
- Follow RESTful conventions

COMMON MIDDLEWARE

express.json()	Parse JSON bodies
express.urlonceded()	Parse URL-encoded bodies
cors	Enable CORS
morgan	HTTP request logger
helmet	Security headers
compression	Compress responses

ENV VARIABLES (.env)

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/myapp
JWT_SECRET=your_jwt_secret
NODE_ENV=production
CLIENT_URL=http://localhost:3000
```

STATUS CODES

200	OK	Success
201	Created	Resource created
400	Bad Request	Invalid request
401	Unauthorized	Authentication required
403	Forbidden	Access denied
404	Not Found	Resource not found
500	Internal Server Error	Server error